Sistemas informáticos
Curso 2005-2006

Máquina virtual de Java multi-aplicación

Beatriz Rodríguez Jiménez
Marta Texidor Méndez de Vigo
David Viñas Domínguez

Dirigido por:
Profesora: Katzalin Olcoz Herrero
Departamento: Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

Alicia se rió: “No tiene sentido intentarlo”, dijo: “No se puede creer en cosas imposibles” “Yo más bien diría que es cuestión de práctica” dijo la reina. “Cuando yo era joven, practicaba todos los días durante media hora. Muchas veces llegué a creer en más de seis cosas imposibles antes del desayuno”

Lewis Carroll, “Alicia en el país de las maravillas”.

ÍNDICE

1 Resumen del proyecto

Español

El objetivo de nuestro proyecto ha sido el de estudiar una máquina virtual de Java, denominada Jikes RVM, y modificarla para que sea multi-aplicación, con la evidente mejora de rendimiento frente a una máquina mono-aplicación.

Actualmente, cada vez que se ejecuta una aplicación en Java, ésta debe usar una máquina virtual propia. Esto significa que si un procesador va a ejecutar varias aplicaciones tiene que cargar e inicializar varias máquinas virtuales, con el consiguiente desperdicio de memoria, tiempo y consumo.

Nuestro enfoque consigue mejorar tiempo y consumo. La mejora en tiempo de ejecución se debe a que se evita cargar la maquina virtual en memoria tantas veces como aplicaciones se quieran ejecutar. La mejora en consumo de memoria se debe a que sólo se tiene cargada en memoria una imagen de la máquina virtual para varias aplicaciones.

La elección de Jikes RVM se debe a que es una máquina virtual de código abierto orientada a la investigación. Por lo tanto su código está diseñado para que sea fácilmente modificado. Por esto, existe mucha documentación relacionada con la máquina virtual. En el campo de las máquinas virtuales, muchos investigadores utilizan Jikes RVM (1).

Además, debido a las características de Jikes RVM, en la cual se van cargando las clases y los métodos que se van necesitando dinámicamente en memoria, para ejecutar dos aplicaciones de forma simultánea sólo se cargarán las clases comunes una vez. Asimismo se disminuye considerablemente el consumo de memoria en caso de tener librerías o paquetes comunes.

English

The aiming of our project has been to study a Java virtual machine, called Jikes RVM, and to modify it so that it turns into a multi-application machine, with the obvious performance improvement compared to a mono-application machine.

Nowadays, every time a Java application is running, it must use its own virtual machine. It means that a processor which is running several applications, it has to load and initialize several virtual machines, with the consequent waste of memory, time and consumption.

Our proposal achieves time and memory improvement. Running time improvement comes from avoiding the load of the virtual machine in memory as many times as applications we want to run. Memory consumption improvement is due to the fact that only one image of the virtual machine has to be loaded for running multiple applications.

Jikes RVM was selected because it is an open-source virtual machine investigation-oriented. So, its code is designed to be easily modified. Moreover, due to this, there is a lot of documentation related with this virtual machine. In the virtual machine area, a lot of researchers use Jikes RVM (1).

Because of the properties of Jikes RVM, which loads classes and methods dynamically “on demand”, when running two applications at the same time the common classes will be loaded only once. This also improves the memory consumption when the applications have libraries or packages in common.

(1) http://jikesrvm.sourceforge.net

2 Breve historia del proyecto Jikes

La introducción del lenguaje de programación Java que se ejecuta sobre una máquina virtual y la popularidad que enseguida obtuvo llevó a IBM, entre otros desarrolladores, a empezar investigaciones significativas en la tecnología asociada a las máquinas virtuales. Dicha tecnología ha sido considerada el año pasado la tecnología que mayor impacto tendrá en el diseño de procesadores en el futuro cercano (*Microprocessor Report*, premio a la mejor tecnología emergente del 2005).

En Noviembre de 1997, un pequeño grupo de investigadores del centro de investigación de IBM inició un proyecto, llamado Jalapeño, cuyos objetivos eran el desarrollo de una infraestructura de investigación para máquinas virtuales.

Esta máquina virtual fue concebida como una forma de evaluar y probar técnicas de implementación de máquinas virtuales (*The Jikes Research Virtual Machine Project*, Alpern et Al; IBM SYSTEMS JOURNAL, VOL 44, 2005).

Jalapeño se basaba, fundamentalmente, en dos grupos de trabajo separados. Uno encargado de la funcionalidad del núcleo de la máquina virtual y otro de optimizar el compilador.

Durante el transcurso del proyecto, muchos de sus miembros publicaron sus resultados en varios foros de investigación. Esta publicidad hizo que la comunidad científica se mostrará interesada en el proyecto. La primera universidad que mostró interés en el proyecto fue la Universidad de Massachusetts.

Aunque inicialmente en IBM no se había pensado en esta posibilidad, después de muchos esfuerzos técnicos y burocráticos se proporcionó el sistema desarrollado a las universidades bajo una licencia diseñada para el caso.

Durante unos años se realizó una gran labor de investigación, desarrollando otros aspectos del proyecto y mejorando su funcionalidad, llevada a cabo tanto por IBM como por varias universidades.

Esta investigación dio como resultado el desarrollo de una máquina virtual con una gran funcionalidad, en la que estaban involucrados numerosos investigadores. Esto llevó a IBM a permitir que Jalapeño se convirtiera en un sistema de código libre contribuyendo a la comunidad de código abierto.

IBM permitió esto porque tenía otros dos sistemas comerciales, por lo que Jalapeño no era una propiedad vital para la compañía y por los beneficios e impacto que este proyecto proporcionaría a la comunidad de investigación.

Fue en este momento, en 2001, cuando se cambió el nombre del proyecto y surgió Jikes RVM, proyecto que todavía sigue en desarrollo y al que contribuyen 85 universidades de todo el mundo, entre ellas la Universidad Complutense de Madrid.

3 Introducción a Jikes RVM y sus principales características

Partiendo de un proyecto interno de investigación de IBM llamado Jalapeño surgió Jikes RVM (Research Virtual Machine) como un proyecto de código abierto en el que han participado a lo largo de los años universidades como la Universidad de Massachusetts en Amherst o la Australian National University.

Jikes RVM ha desarrollado software con la intención de ser usado como una "caja blanca". No con la idea de ofrecer una gran funcionalidad, sino como un sistema en el que lo interesante es su diseño interno. Este hecho es el que permite la investigación, principal objetivo que tiene este proyecto.

Jikes RVM es una máquina virtual para Java escrita en el lenguaje de programación Java con una pequeña parte escrita en C. Esto hace que una de las principales características de Jikes RVM es que se ejecuta a sí misma, esto quiere decir que su código Java se ejecuta sin necesidad de requerir una segunda máquina virtual. Sin embargo, es necesario tener instalada otra máquina virtual que se utiliza en el proceso de compilación de Jikes RVM, aunque este paso sólo es necesario realizarlo una vez.

La principal motivación que llevó al diseño de una máquina virtual escrita en Java fue el hecho de probar si era posible hacerlo. El desafío de usar un lenguaje de programación moderno, orientado a objetos, con tipos seguros y sobre todo con recolección automática de basura era considerable.

Jikes RVM ha sido diseñada para ser lo más autosuficiente posible. El modelo de objetos y la disposición de la memoria, que explicaremos detalladamente más adelante, permite un rápido acceso tanto a las clases como a los atributos y los métodos de éstas.

Aunque inicialmente se intentó diseñar una máquina virtual portable, este objetivo no se ha cumplido hasta el momento y Jikes RVM no es una máquina virtual portable ya que sólo puede ejecutarse en plataformas AIX/PowerPC, Linux/PowerPC y Linux/IA-32. Es esta última plataforma la que hemos utilizado para la realización de nuestro proyecto. El motivo de que la máquina virtual no sea portable se debe a que la disposición de los objetos y los mecanismos de bloqueo son específicos de cada arquitectura. Sin embargo, ahora mismo hay proyectos en desarrollo que se dedican a conseguir que Jikes RVM sea tan portable como sea posible.

La parte de la máquina virtual escrita en C se debe a la necesidad de usar las llamadas del sistema operativo para tener acceso al sistema de ficheros y a los recursos del procesador. Se utilizó para ello la librería estándar de C y por ello una pequeña parte de Jikes RVM está escrita en este lenguaje de programación.

Para tener una idea general de Jikes RVM podemos considerar que se divide principalmente en los siguientes componentes:

- **El núcleo de ejecución.**
  Esta formado por el hilo planificador, el cargador de clases, las librerías que dan soporte a la máquina etc... Estos componentes son responsables de la gestión de las estructuras de datos necesarias para la ejecución de aplicaciones y de la comunicación con las librerías.

- **Los compiladores.**
  Esta formado por el compilador básico y los optimizadores. Este componente es el responsable de generar código máquina ejecutable a partir del código byte.

- **Los gestores de memoria.**
  Este componente se encarga de la ubicación y recolección de los objetos generados durante la ejecución de las aplicaciones.

- **Sistema de optimización adaptativo.**
  Este componente se responsabiliza de perfilar la ejecución de una aplicación usando un compilador optimizado para mejorar, todo lo que sea posible, su rendimiento.

4 Las estructuras de datos más relevantes en Jikes RVM

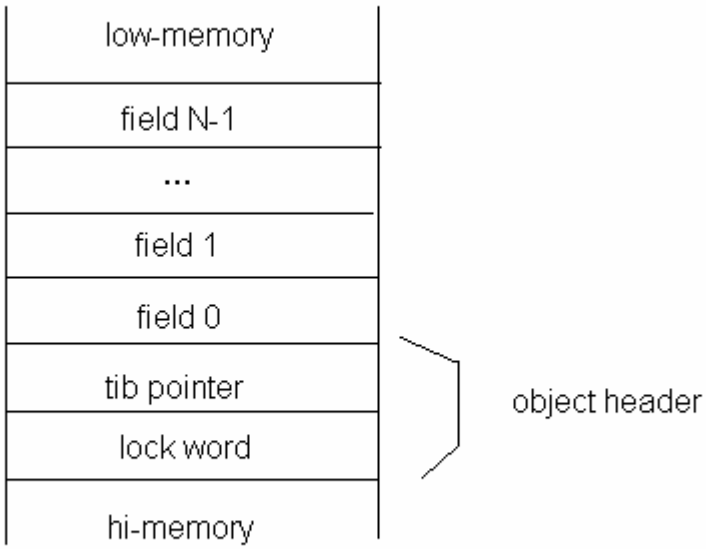
En este apartado vamos a abordar la descripción de las estructuras de datos más importantes que mantiene Jikes RVM durante su ejecución, tanto a nivel funcional como de implementación.

4.1 La imagen de la máquina virtual

La imagen de la máquina virtual Jikes RVM está formada totalmente por objetos Java.

Uno de estos objetos Java que conforman la máquina virtual es el denominado *the boot record* que está implementado por la clase *VM_BootRecord.java* y que es el área de comunicación entre el sistema operativo y la máquina virtual.
La información que contiene esta clase es necesaria para empezar la ejecución de la máquina virtual y por lo tanto la creación de la instancia que se hace de esta clase se realiza en tiempo de compilación cuando se construye Jikes RVM.

The boot record tiene la siguiente apariencia:



Algunos de sus campos son:

- El *spRegister* que apunta a la pila de la máquina virtual. Este objeto pila es un array de bytes.

- El *ipRegister* que apunta a la primera palabra de un array de instrucciones máquina. Estas instrucciones son en el código que debe ejecutar la máquina virtual cuando va a empezar a ejecutarse, es decir el código de *VM.boot().*

- El *tocRegister* es el campo que apunta a un array de palabras que contiene las direcciones de los atributos y métodos estáticos de la imagen de la máquina virtual, el JTOC *(Jikes table of contents).* Debido a la importancia de esta estructura, le dedicaremos un apartado más adelante.

- Las direcciones de comienzo y fin de la máquina virtual dentro de la memoria.

- La información relacionada con los heaps de la máquina virtual como son el tamaño máximo de heap o los rangos de direcciones donde están almacenados los heaps que usará la máquina virtual. Es en estos heaps donde se guardan las instancias de los objetos.

Por lo tanto el esquema de la máquina virtual es de la siguiente forma:

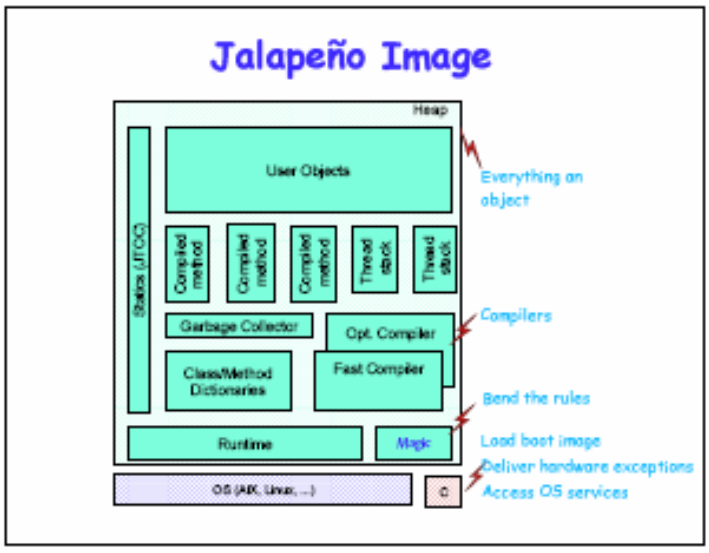


Figura 4-1: Esquema de la máquina virtual.

4.2 Los objetos Java en Jikes

Para Jikes RVM, un objeto puede ser o bien un array o bien un objeto simple. Cada uno se representa en memoria de forma diferente, pero la base es la misma: ambos tipos se representan mediante un array en el más puro sentido de la palabra, es decir, una serie de posiciones de memoria consecutivas, cada una de 4 bytes. Dos posiciones por debajo de la dirección de referencia al objeto comienza la denominada cabecera del objeto.

La cabecera del objeto está formada por dos posiciones de memoria. La primera es la palabra de estatus o estado (*status word*), la cual se divide en tres campos binarios:

- El primero se utiliza para funciones de bloqueo y cerrojos del objeto.
- El segundo guarda el valor hash predeterminado para los objetos que puedan someterse a una función hash.
- El tercero es utilizado por el sistema de gestión de memoria de Jikes.

La longitud de estos campos se determina en tiempo de compilación mediante una serie de constantes.

La segunda posición guarda la dirección del TIB (*Type Information Block*) correspondiente al objeto. Del TIB hablaremos más adelante; por ahora, para hacernos una idea, sólo diremos que hace referencia a la clase a la que pertenece el objeto.

Si se trata de un array de objetos, entonces en la dirección de referencia nos encontramos con el primer elemento del array, y el resto de elementos se van posicionando en sentido ascendente de las direcciones de memoria. Cada elemento será la referencia al objeto ubicado en dicha posición del array, si se trata de un array de objetos, o bien directamente el valor guardado en dicha posición, si es un array de tipos primitivos.

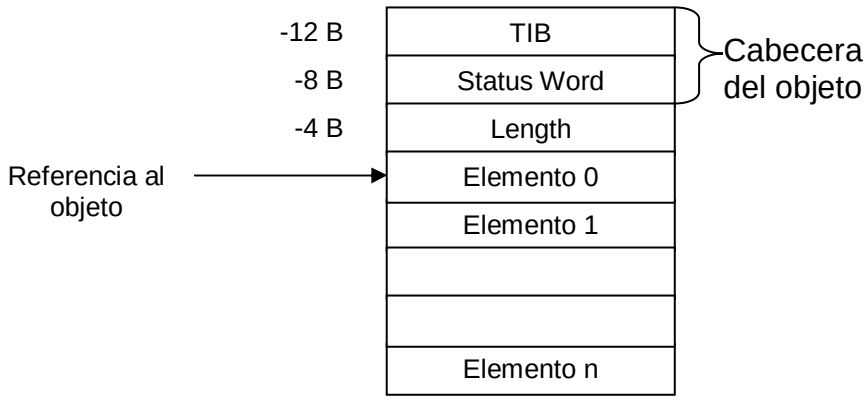


Figura 4-2: Representación en memoria de un objeto

Si se trata de un objeto simple (*scalar object*) entonces a partir de la cabecera del objeto y en sentido descendente se van ubicando los distintos atributos del objeto, que de nuevo serán o bien referencias a objetos o bien valores primitivos. Con lo cual, en el caso de objetos simples, tenemos que la posición apuntada por la referencia al objeto (el elemento 0 en el caso de los arrays) y la posición anterior (la longitud en el caso de los arrays) no se utilizan.

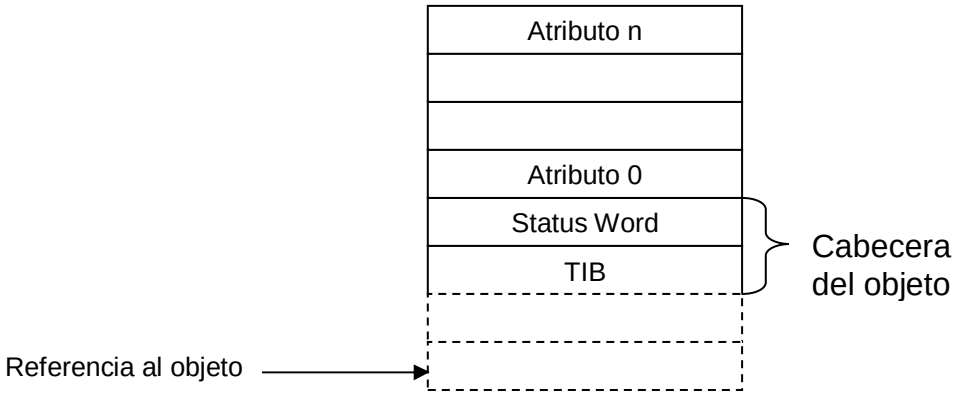


Figura 4-3: Representación de un objeto Java en Jikes

4.3 El TIB y los métodos compilados

El TIB (*Type Information Block*, o bloque de información del tipo) es una estructura que almacena la información que define una clase Java. En memoria se representa igual que un array de objetos Java (Object [ ]). El primer elemento es una referencia a una estructura que contiene información acerca de la clase. Por ejemplo, su superclase, los interfaces que implementa o información acerca de sus métodos. El resto de los elementos son referencias a los métodos de la clase.

Cada método se almacena compilado en código máquina en forma de un array de enteros int [ ].

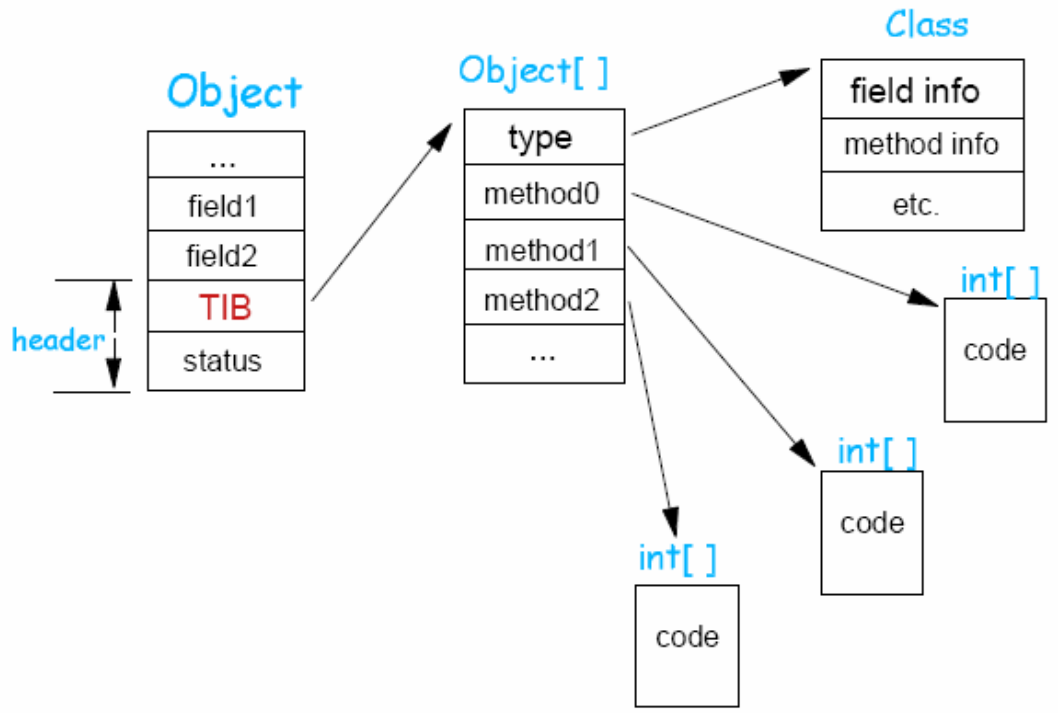


Figura 4-4: Estructura general de un TIB

4.4 El JTOC (Jalapeño Table Of Contents)

El JTOC se implementa en la clase VM_Statics.java.

Tiene la misma estructura que un arrray de enteros int [ ] (Ver figura 4-5). En él se guardan:

- Referencias a todos los TIB de las clases cargadas, tanto de la aplicación como de la propia Jikes RVM.
- Referencias a todos los métodos estáticos.
- Todos los atributos estáticos (su valor en el caso de tipos primitivos o su referencia en el caso de objetos).
- Constantes numéricas.
- Referencias a constantes del tipo String.

Esta estructura permite realizar de forma rápida comprobaciones de tipos en tiempo de ejecución, acceso a constantes y acceso a miembros estáticos.

En realidad no es un auténtico objeto Java, pues aunque se declara como un array de enteros guarda elementos de diferentes tipos. La referencia al JTOC está permanentemente almacenada en un registro denominado *JTOC Register.* El primer elemento del JTOC apunta a este registro. El valor de este puntero, y por lo tanto la posición de memoria en la que se va a cargar el JTOC se calcula en el momento de construir la máquina.

Paralelo al JTOC tenemos una estructura declarada como array de bytes (byte [ ]) denominada *JTOC Descriptor.* Esta estructura guarda en la posición i-ésima el tipo de información almacenada en el i-ésimo elemento del JTOC.

Todas las estructuras globales de datos de Jikes RVM son accesibles a través del JTOC. Para permitir un chequeo de tipos dinámico y rápido, el JTOC contiene referencias al TIB de cada clase del sistema.

Una de las ventajas que ha proporcionado el uso del lenguaje de programación Java ha sido la posibilidad de cargar clases, e incluso métodos, durante la ejecución de una aplicación.

Cuando el compilador de Jikes RVM encuentra código byte que hace referencia a una clase que todavía no ha sido cargada, no carga la clase inmediatamente. El compilador emite código que cuando se ejecuta por primera vez se asegura de que la clase referenciada ha sido cargada y después ejecuta la operación.

Inicialmente, pensamos en la posibilidad de duplicar el JTOC, de tal manera que podríamos separar las clases de cada una de las diferentes aplicaciones a ejecutar y de las clases de la máquina virtual entre sí.

Sin embargo, en el JTOC no se guardan las instancias de dichas clases sino solamente su código compilado y la información acerca del tamaño que ocupan los objetos de esas clases, por lo que llegamos a la conclusión de que no obtendríamos un mayor rendimiento si implementáramos varios JTOC, ya que las clases que se usan en común en varias aplicaciones, como por ejemplo las del paquete *java.lang*, sólo se cargan una vez. También ocurre esto en el caso de se ejecuten dos aplicaciones iguales a la vez (las clases de la aplicación sólo se cargarán una vez). Por lo tanto, en estos dos casos está propiedad del JTOC permite un ahorro en el espacio de memoria y tiempo de carga, que es el objetivo que buscamos.

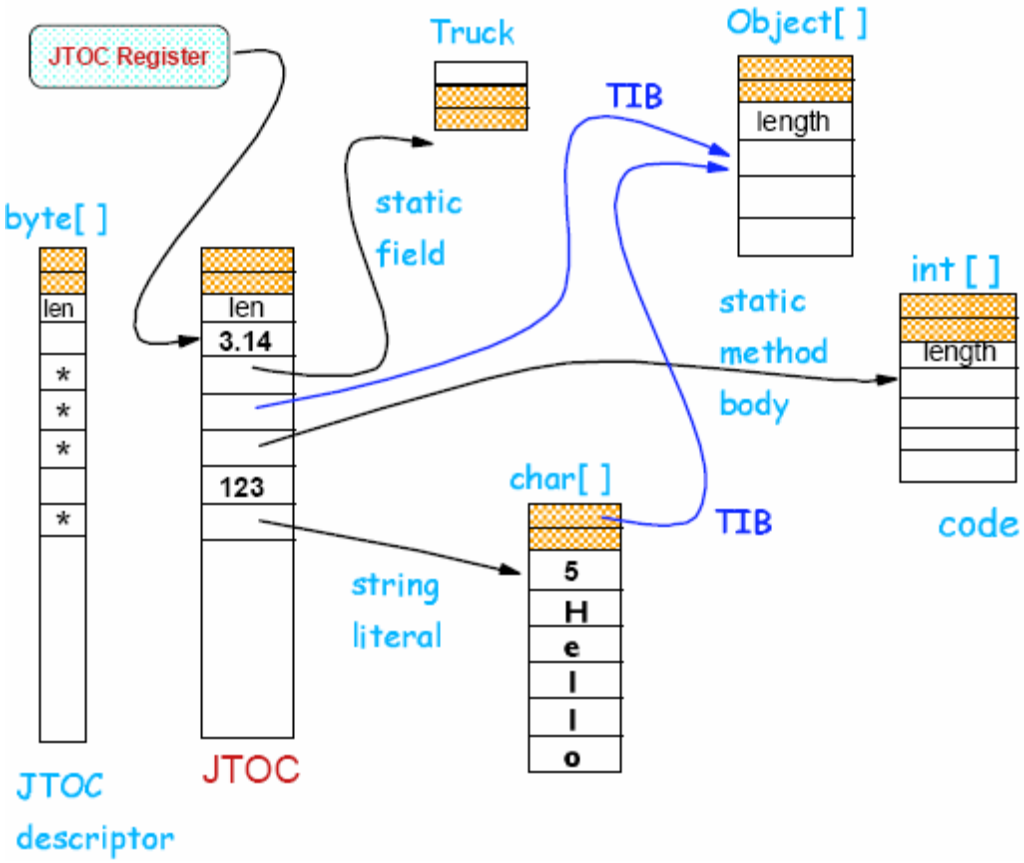


Figura 4-5: Ejemplo del JTOC y estructuras relacionadas.

Sólo hay un único JTOC durante la ejecución de Jikes RVM (y un único *JTOC Register* asociado).

4.5 La pila de métodos y el marco de pila (Stack Frame)

Toda aplicación ejecutada sobre Jikes RVM se lanza como un hilo (más de uno si la aplicación es multihilo). Dicho hilo inicial se denomina hilo principal (*MainThread*) y su ejecución arranca con el método main de la aplicación. Para gestionar hilos Jikes RVM mantiene una pila de métodos para cada uno. Dicha pila estará formada por los denominados Marcos de Pila.

Por cada método que se esté ejecutando en un hilo tendremos un marco de pila en la pila asociada al hilo. Cada vez que aparezca la invocación a un método se generará un nuevo marco de pila asociado a dicho método y se apilará en la pila correspondiente. Una vez terminada la ejecución de un método se desapila su marco y se continúa la ejecución del método que lo invocó, cuyo marco de pila es ahora la cima.

La estructura de un Stack Frame la podemos ver en la siguiente figura:

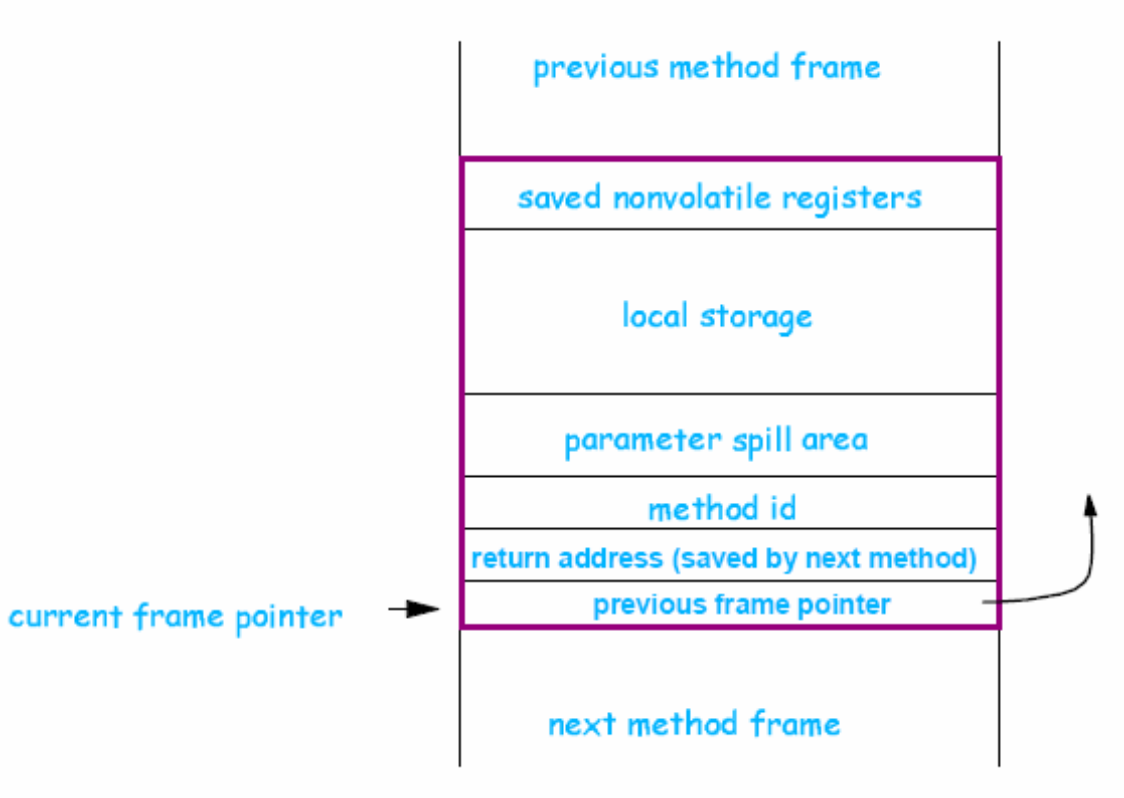


Figura 4-6: Estructura de un Stack Frame.

El puntero al marco anterior (*previous frame pointer*) nos permite hacer un desapilado rápido simplemente cambiando el *current frame pointer*. La dirección de retorno (*return address*) indica en qué instrucción del método nos quedamos cuando cambiamos a un nuevo Stack Frame. El campo *method id* contiene la información necesaria para localizar el método que estamos ejecutando (recordemos que dicho método estará guardado en forma de int [ ] y referenciado desde el TIB de la clase a la que pertenezca).

La zona destinada a la ubicación de parámetros (*parameter spill area*) será donde se almacenen los datos asociados a los parámetros. Su tamaño depende del tipo y cantidad de parámetros del método, pudiendo incluso no existir en caso de tratarse de un método sin parámetros. La zona de almacenamiento es la destinada a las variables locales del método, ya sean tipos primitivos o referencias a objetos.

Un Stack Frame se define como un array de enteros, pero en realidad no cumple las restricciones necesarias para ser un objeto Java pues almacena diversos tipos de elementos. El JTOC y el Stack Frame son las dos únicas estructuras de datos de Jikes RVM que no cumplen las condiciones de Java.

4.6 Los montículos o heaps

Son las zonas donde realmente se almacenan elementos como los TIB´s, los métodos compilados o los objetos Java de la aplicación y de Jikes RVM. El número de heaps puede modificarse manualmente desde línea de comandos al lanzar a ejecución Jikes RVM. El número de heaps que se crean por defecto es veinte.

5 El modelo de objetos en Jikes RVM y el cargador de clases

5.1 El modelo de objetos

Los tipos en Java tienen tres formas: primitivos, clases y arrays. Las clases pueden ser clases propiamente dichas o interfaces. Jikes RVM representa los tipos de Java como objetos de la clase *VM_Type*. Esta clase tiene tres subclases: *VM_Primitive* para la representación de tipos primitivos, *VM_Class* para la representación de las clases y *VM_Array* para representar los arrays.

Un atributo de la clase *VM_Class* se utiliza para distinguir las clases de los interfaces.

Jikes RVM mantiene también lo que llaman TIB (bloque de información del tipo).

5.2 El cargador de clases

Todas las clases que se utilizan, tanto las de la máquina virtual como las de las aplicaciones que se van a ejecutar deben cargarse en memoria en la estructura JTOC.

A continuación vamos a explicar como se cargan las clases, los arrays y los tipos primitivos en el JTOC. Estas tres son las estructuras que se pueden cargar en el JTOC puesto que se corresponden con los tres tipos de instancias que existen en Java.

Con las tres subclases de *VM_Type* se distinguen los tres tipos de elementos que se pueden guardar en el JTOC.

Puesto que las clases son el principal elemento que se guarda en el JTOC vamos a ilustrar el cargador de clases, teniendo en cuenta que los *VM_Array* y los *VM_Primitive* se cargan de forma similar.

Para cargar una clase lo primero que hay que hacer es generar o buscar (en el caso de que la clase ya haya sido cargada en el JTOC) el nombre con el que se va a identificar a la clase. Este nombre es de tipo *VM_Atom*.

Una vez que tenemos el identificador de la clase hay que buscar o crear su *VM_TypeReference* que es la referencia en un archivo "*.class" a algún tipo que puede ser una clase, un tipo primitivo o un array.

Después, una vez que se tiene el *typeReference*, se devuelve el tipo del objeto que se quiere cargar en la clase que puede ser una clase, un array o un tipo primitivo, como hemos visto anteriormente y en este caso como sabemos que es una clase transformamos el *VM_Type* en una *VM_Class*.

Una *VM_Class* se construye en cuatro fases:

- En la fase de carga, se lee el archivo ".class" pero no se intenta todavía examinar ninguna de las referencias simbólicas que se presentan en él.

- En la fase de *"resolve"* se siguen las referencias simbólicas para ver cuales son las posibles superclases de la clase que nos ocupa y el tamaño de sus atributos y de esta forma se calcula el espacio y la información de desplazamiento para sus atributos. Después, y una vez calculados estos valores, se reserva espacio en el JTOC para los atributos estáticos, los métodos estáticos y la tabla de métodos virtuales de la clase.

- En la fase "*instantiate*" se compilan los métodos de la clase de código byte a código máquina (que es la información que se guarda en el JTOC), se construye la información del TIB (*Type Information Block*) de la clase y se rellena el JTOC con esta información.
  Por lo tanto en el JTOC se guarda la siguiente información de las clases cargadas: Sus atributos estáticos, sus métodos estáticos y el código máquina de sus atributos, es decir, la estructura de las clases pero no las instancias de los objetos de las clases.

- La fase "*initialize*" ejecuta el inicializador estático de la clase, que no tiene porque existir siempre (suelen ejecutarse los métodos init () de las clases).

Los *VM_Array* se construyen de una forma similar y en los tipos *VM_Primitive* las fases de "*resolution*"," *instantiation*" y "*initialization*" no realizan ninguna operación.

6 Compilación y construcción de Jikes RVM

Muy poco de Jikes RVM no se escribe en Java. La máquina virtual de Jikes se diseñó para funcionar como un proceso de usuario de AIX. Como tal, debe utilizar el sistema operativo para tener acceso al sistema de ficheros, a la red y a los recursos subyacentes del procesador. Para tener acceso a estos recursos, existían dos opciones: el núcleo de AIX podía llamarse directamente usando las convenciones para realizar llamadas a procedimientos y funciones de bajo nivel, o podía ser accedido a través de la librería estándar de C. Se eligió la última alternativa para evitar las dependencias específicas del núcleo del sistema operativo. Esto provocó que una pequeña porción de Jikes RVM esté escrita en código C en lugar de Java.

Hasta la fecha, la cantidad de código de C requerida ha sido pequeña (cerca de 1000 líneas). Alrededor de la mitad de este código consiste en las funciones simples que transmiten llamadas entre los métodos de Java y la librería estándar de C. El único propósito de este código es el de convertir parámetros y valores entre el formato de C y el formato de Java. La otra mitad del código de C consiste en un cargador del "cargador" de la máquina y dos gestores de señales. El cargador del cargador asigna la memoria para la imagen virtual de la máquina, lee la imagen del disco en memoria, y permite el acceso al código de lanzamiento de la imagen. Los gestores de señales se encargan de capturar las interrupciones hardware y pasar las interrupciones de reloj del sistema a Jikes RVM.

Un sistema de servicios fundamentales (un cargador de clases, un asignador de memoria de objetos, un compilador) debe existir antes de que una JVM pueda cargar todos los servicios restantes requeridos para su ejecución normal. Los servicios iniciales para una JVM escrita en código nativo, o una JVM que funciona sobre otra JVM, están disponibles a través de una rutina de servicio subyacente. Jikes RVM no está escrito en código nativo y no tiene ninguna rutina de servicio subyacente. Por lo tanto, se montan los servicios esenciales de la base en una imagen ejecutable del cargador (*boot-image*) antes de ejecutar la JVM. Esta imagen es escrita en un archivo. Más adelante, este archivo es cargado en memoria y ejecutado.

El *boot image* es creado por un programa de Java llamado *boot-image writer*. Éste construye la simulación de una máquina virtual corriente de Jikes RVM y después lo empaqueta en una imagen del cargador (*boot image*). El *boot image writer* es un programa corriente de Java y puede funcionar en cualquier JVM. La JVM que ejecuta el *boot image writer* se suele llamar *source JVM*, y la máquina virtual que se obtiene se llama *target JVM.*

El *boot image writer* se asemeja a un compilador y a un enlazador: convierte código byte a código máquina y reescribe direcciones físicas de memoria para convertir los componentes del programa en una imagen ejecutable. Sin embargo, como los compiladores de Jikes, los cargadores de la clases y las estructuras de datos en ejecución están todos en código Java, y en

esto se diferencia de la mayoría de los compiladores, en el *boot image writer* también deberán encontrarse objetos “vivos”.

El *boot image writer*, en la *source JVM,* instancia objetos Java que representan la *target JVM*. Entonces utiliza la facilidad incorporada de la reflexión de Java para traducir estos objetos construidos con el modelo de objetos de la simulación a modelos de objetos de Jikes RVM. Esta forma del *boot image writer* de referirse a sí mismo es relativamente sencilla: es un traductor de modelos de objetos.

Puesto que Jikes RVM es un programa en Java, cada uno de sus componentes es un objeto de Java y el *boot image writer* puede construir la simulación ejecutando métodos init () especiales en cada uno de los subsistemas importantes de Jikes. Un cargador de clases modificado para requisitos particulares se cerciora de que cualquier clase necesaria para ejecutar este código está cargada en la simulación así como en la *source JVM*. Mientras se carga una clase, se compilan (en un compilador que se ejecuta en la *source JVM*) y se incluyen sus métodos en la simulación.

Esta estrategia de cargar las clases tanto en la *source JVM* como en la simulación de la *target JVM* requiere una lista completa de las clases para tener éxito. Si, cuando Jikes RVM comienza su ejecución, un método del entorno de ejecución se refiere a cualquier clase que no se encuentra en la imagen del cargador (*boot image*), se produciría una recursión infinita: el entorno de ejecución necesita cargar parte de sí mismo para cargar otra parte de sí mismo...y así sucesivamente.

El problema de determinar el sistema mínimo de clases necesarias en la simulación para prevenir esto fue solucionado usando una combinación de planificación y ensayo y error cuidadosos. Todas las clases de la base de Jikes fueron nombradas con un prefijo VM_. Éstas son las clases necesarias para proporcionar la maquinaria suficiente para permitir que la máquina virtual realice la compilación, la gestión de memoria, y la carga dinámica de clases. El prefijo especial es reconocido por los compiladores de Jikes y utilizado para suprimir reglas de enlazado dinámico normales: nunca generan código de enlazado dinámico entre los métodos de las clases que tienen este prefijo. Las clases de la base también fueron escritas cuidadosamente para evitar el uso innecesario de las clases de la biblioteca de Java. Las clases fundamentales *java.lang.Object, java.lang.Class, java.lang.String* y algunas clases de entrada-salida eran excepciones inevitables. Unidas, las clases de VM_ y las clases fundamentales de Java forman un sistema inicial de clases que necesariamente deben aparecer en el *boot image*.

Una pequeña cantidad de dependencias adicionales (por ejemplo, Integer, Float, Double, y varias clases de arrays y excepciones) fueron identificadas por ensayo y error. Se construyó un *boot image* y se intentó ejecutar la máquina. Si falló (recurrentemente) intentando cargar la clase X, se agregó dicha clase a la lista de las clases escritas en el *boot image* y se repitió el proceso.

Cuando la simulación se completa, se transforma en un *boot image*. Esto implica el encontrar todos los objetos en la simulación, el convertirlos al formato de los objetos de Jikes RVM y el almacenarlos en un array del *boot image*. Todos los componentes de una máquina virtual Jikes en ejecución se pueden alcanzar utilizando un único array: JTOC (véase apartado 4.4). En la simulación, el JTOC es codificado por tres arrays paralelos: un array de enteros (para los valores primitivos), un array de instancias de objetos (para las referencias), y un array de booleanos para discriminar entre los dos. La estructura generada en el JTOC se recorre recursivamente y sus valores, tanto las referencias como los primitivos, son encontrados, traducidos y guardados en el array del *boot image*. Puesto que el TIB, bloque de información del tipo, (véase apartado 4.3), para cada clase cargada es referenciado desde el JTOC, todos los cuerpos de los métodos compilados necesarios serán incluidos en la imagen del cargador (*boot image*).

El proceso de traducción utiliza la reflexión. En la simulación, el *boot image writer* obtiene el objeto *java.lang.Class* para cada objeto y obtiene todos sus campos iterando sobre él con el método *getFields ()*. Para cada campo, extrae su valor en el objeto *source* y extrae el desplazamiento de ese mismo campo en el *target* de Jikes RVM desde la descripción de la clase para crear el objeto correspondiente. Entonces, escribe el valor en ese desplazamiento en el *boot image*. Cuando se encuentran referencias a un objeto, no podemos utilizar ningún valor de la simulación. Las referencias en la simulación se convierten en direcciones del *boot image* usando una tabla hash que se asigna en el espacio donde está ubicado el *boot image*. (Un array que contiene las direcciones de todas las referencias en el *boot image* se puede incluir en la imagen del cargador para apoyar la reubicación de ésta en tiempo de carga.)

El *boot image writer* copia los objetos de Java, campo por campo, de la simulación al *boot image*, traduciendo simultáneamente desde el *source JVM* al modelo de objeto del *target JVM*.

Además de los objetos accesibles desde el JTOC, otros dos objetos se necesitan en el *boot image writer*: un hilo inicial que contiene una pila vacía listo para ejecutar la primera instrucción del método boot() cuando Jikes comienza su ejecución y *boot record* que interconecta el *boot image* con el *boot image runner* (descrito después). Este *boot record* contiene el comienzo, el final, y las últimas direcciones usadas en la imagen, cuatro valores de registros usados para comenzar la ejecución de Jikes RVM, la dirección del método boot(), y de las direcciones para las llamadas del sistema AIX. Cuando estos valores están almacenados en el *boot image array*, se escriben en el disco.

Un programa corto llamado *boot image runner* comienza la ejecución de Jikes. Lee el *boot image* en memoria, fija los cuatro registros a los valores indicados, y accede al método boot(). El *boot image runner* está escrito en C (con una pequeña parte en código ensamblador para fijar los registros), no es código Java, así que no requiere tener una JVM funcionando.

Cuando el método boot() comienza a ejecutarse, la máquina virtual está en un estado frágil: puede ejecutar un solo hilo de instrucciones máquina, pero

todavía no ha creado los recursos externos del sistema operativo que necesita para apoyar su propia ejecución. Estos recursos del sistema operativo no se pueden crear por el *boot image writer*, porque se refieren al estado externo que no existirá hasta que se ejecute el *boot image*. Así, Jikes RVM debe realizar una inicialización adicional.

En el tiempo de carga, la máquina virtual inicializa direcciones hardware específicas, abre los archivos que se corresponden con los objetos de *System.in*, de *System.out*, y de *System.error* de la librería de Java, analiza los argumentos de la línea de comandos, y crea un objeto de *System.Properties* que se corresponde con el entorno actual de ejecución. Entonces, el subsistema multihilo es inicializado creando los hilos del sistema operativo que se utilizarán como procesadores virtuales sobre los cuales se multiplexan los hilos de Java.

Jikes RVM se ejecuta hasta que el último hilo Java (que no sea un demonio) termina o se llama a *System.exit ().*

7 Lanzamiento a ejecución de aplicaciones

Empezamos por estudiar la clase *VM.java* encargada de la inicialización de la máquina virtual.

Primero se compila y se construye la máquina virtual (veáse el apartado anterior). Esto se hace en tiempo de compilación con la configuración que se indique y sólo es necesario hacerlo una vez si no se quiere modificar Jikes RVM.

Una vez construida la máquina virtual se tiene que cargar en memoria virtual. Esta parte de la máquina no hemos necesitado modificarla.

Una vez cargada la máquina e inicializados todos los sistemas que necesita para su funcionamiento; como el recolector de basura, el gestor de memoria o el planificador de hilos; la máquina virtual está lista para ejecutar una aplicación. En el caso de la máquina original sólo se puede ejecutar una aplicación, nosotros hemos tenido que modificar esta parte de la máquina para que puedan ejecutarse varias.

La ejecución de aplicaciones Java en Jikes RVM se basa en la creación y ejecución de un hilo Java. Para crear estos hilos la máquina virtual utiliza la clase *VM_Thread*, que implementa el contexto de ejecución de un hilo Java en Jikes RVM.

A continuación mostramos como realiza esta tarea la máquina original y después las modificaciones que hemos realizado para que la máquina sea multi-aplicación.

Lo primero que hace la máquina virtual es procesar la línea de comandos, que originalmente está preparada para tratar solamente una aplicación. Para ello la máquina virtual utiliza la función
VM_CommandLineArgs.lateProcessCommandLineArguments( )
que devuelve como un *applicationArguments String[ ]* los argumentos que tiene la aplicación, el primero de los cuales es la clase que se debe ejecutar.

A continuación se crea un *MainThread* que será el hilo principal que ejecutará la aplicación, de la siguiente forma:

```
Thread      xx         = new MainThread(applicationArguments);
VM_Address  yy         = VM_Magic.objectAsAddress(xx);
VM_Thread   mainThread =(VM_Thread)VM_Magic.addressAsObject(yy);
```

Figura 7-1: Lanzamiento de una aplicación en la máquina original

Se crea un objeto de la clase *MainThread* (un hilo *java.lang.Thread*), a cuyo constructor se le pasan los argumentos que necesita la aplicación para ser ejecutada.

Posteriormente, se obtiene la dirección *VM_Address* donde ha sido cargado el hilo y después este hilo de Java se transforma en un *VM_Thread*.

De esta forma se ha creado el hilo que va a ejecutar la aplicación.

Para que la máquina virtual pueda ejecutar varias aplicaciones ha sido necesario modificar este código sustituyéndolo por el que se muestra en la figura 7-2. Este código realiza las mismas operaciones que el de la máquina virtual original pero para cada una de las aplicaciones que se vayan a ejecutar. Hay que tener en cuenta que debemos realizar un tratamiento de *applicationArguments* para separar las diferentes aplicaciones que se vayan a ejecutar y sus argumentos de entrada.

Para separar las diferentes aplicaciones hemos elegido como elemento la coma.

Además debemos tener en cuenta que necesitamos distinguir unas aplicaciones de otras para lo cual hemos optado por utilizar un identificador de tipo entero en los hilos, de tal manera que distinguimos la aplicación a la que pertenece cada hilo. Utilizamos el identificador 0 para los hilos de la máquina virtual y de 1 en adelante para las aplicaciones que tienen que ejecutarse.

```
//línea de comandos sin las opciones de la máquina virtual
String[] applicationArguments =
VM_CommandLineArgs.lateProcessCommandLineArguments();

//array con las aplicaciones que se van a ejecutar
VM_Thread[] aplicaciones=new VM_Thread [2];

//contador del número de aplicaciones
int numAplicaciones=0;

//variables auxiliares
int j=0;
int r;

//array que guarda los argumentos para cada una de las
//aplicaciones que van a ejecutarse
String [][] matrizArgumentos =new String [2][20];

while (j<applicationArguments.length){
   r=j;
   String[] argumentosAux;
   int contador=0;
   while((r<applicationArguments.length)&&
         (!applicationArguments[r].equals(","))){
          matrizArgumentos[numAplicaciones][contador]
                                    =applicationArguments[r];
          contador++;
          r++;
  }

  if(r<applicationArguments.length) r++; //para saltarme la coma

  //genero en argumentosAux los argumentos de la aplicación que
  //voy a ejecutar
  argumentosAux=new String[contador];

  for(int i=0; i<contador;i++){

     argumentosAux[i]=matrizArgumentos[numAplicaciones][i];
  }
```

```
  //creo el nuevo hilo para la aplicación
  Thread      xx         = new MainThread(argumentosAux);
  VM_Address  yy         = VM_Magic.objectAsAddress(xx);
  VM_Thread   mainThread
=(VM_Thread)VM_Magic.addressAsObject(yy);

 //pongo el identificador de la aplicación al hilo
 mainThread.setIdAplic(numAplicaciones+1);

aplicaciones[numAplicaciones]=mainThread;
numAplicaciones++;
j=r;
}
```

Figura 7-2: Tratamiento de la línea de comandos y creación de las nuevas aplicaciones.

Una vez creados los hilos principales que se van a ejecutar, tenemos que lanzarlos a ejecución.

Para ello utilizamos el método *start()* que debemos invocar para cada aplicación como se muestra a continuación:

```
for(int i=0;i<numAplicaciones;i++){
     aplicaciones[i].start();
}
```

Figura 7-3: Lanzamiento de múltiples aplicaciones.

Este método comenzará la ejecución del hilo principal correspondiente en el procesador virtual adecuado (en nuestro caso sólo hemos utilizado uno), y por lo tanto de la aplicación incluyéndolo en el planificador de la máquina virtual y notificando a éste su comienzo.

8 Ejecución de hilos en Jikes RVM

Al crear un hilo Java se genera también una pila para ese hilo. En el caso de Jikes RVM esta pila se guarda como un atributo privado de tipo byte[ ], perteneciente a la clase *VM_Thread*.

El hecho que el propio hilo cree su pila en su constructor y que lo guarde como un atributo privado nos proporciona la seguridad de que todas las pilas de los hilos de nuestras distintas aplicaciones son privadas y están protegidas, es decir que no se producirán accesos ni modificaciones no deseadas. Por lo que las pilas de nuestras aplicaciones estarán separadas y protegidas unas de otras.

El planificador de la máquina virtual ejecuta el hilo a través del método *run()* de la clase *MainThread* mostrado en la figura 8-1 y cuyo funcionamiento detallamos a continuación.

Lo primero que hace este método es cargar en la estructura JTOC la clase que se tiene que ejecutar. Para más detalles consultar el apartado sobre el cargador de clases.

Una vez creada la estructura *VM_Class*, donde se guarda la información relacionada con la clase, y cargada en el JTOC hay que buscar el método *main()* para ejecutarlo. En el caso de que no se encontrara este método y teniendo en cuenta que estamos hablando de un *MainThread* se produciría un error en la máquina virtual.

A continuación se compila el método principal a código máquina y finalmente se invoca el método para ejecutarlo pasándole como parámetro los argumentos de las aplicaciones y las instrucciones máquina del método que deben ser ejecutadas.

```
public void run () {

      MM_Interface.startGCSpyServer();

    // Set up application class loader
    ClassLoader cl = VM_ClassLoader.getApplicationClassLoader();
    setContextClassLoader(cl);

    // find method to run
    // load class specified by args[0]
    VM_Class cls = null;
    try {
      VM_Atom mainAtom =
VM_Atom.findOrCreateUnicodeAtom(args[0].replace('.','/'));
      VM_TypeReference mainClass =
VM_TypeReference.findOrCreate(cl,
mainAtom.descriptorFromClassName());

      cls = mainClass.resolve().asClass();
      cls.resolve();
      cls.instantiate();
      cls.initialize();
    } catch (NoClassDefFoundError e) {
      // no such class
      VM.sysWrite(e+"\n");
```

```
      return;
    }

    // find "main" method

    mainMethod = cls.findMainMethod();
    if (mainMethod == null) {
      // no such method
      VM.sysWrite(cls + " doesn't have a \"public static void
main(String[])\" method to execute\n");
      return;
    }

    // create "main" argument list

    String[] mainArgs = new String[args.length - 1];
    for (int i = 0, n = mainArgs.length; i < n; ++i)
      mainArgs[i] = args[i + 1];

    mainMethod.compile();

    // Notify other clients that the startup is complete.

    VM_Callbacks.notifyStartup();

    // dummy call for debugger to find the main method
    // (needed for the default option of stopping in the main
method on start up)
    VM.debugBreakpoint();

    launched = true;
    // invoke "main" method with argument list
    VM_Magic.invokeMain(mainArgs,
mainMethod.getCurrentCompiledMethod().getInstructions());
  }
```

Figura 8-1: Método run() de la clase MainThread.

9 El planificador de hilos en Jikes

La clase *VM_Scheduler* implementa el planificador de hilos que utiliza Jikes RVM.

El planificador de hilos de la máquina coordina los hilos de las aplicaciones Java y los hilos demonios de Jikes RVM dentro de un procesador virtual, para que se puedan ejecutar simultáneamente.

El planificador de Jikes RVM de la máquina virtual original es de tiempo compartido, de tal manera que se pueden ejecutar sin problemas aplicaciones multi-hilo.

Debido a eso no ha sido necesario que modificáramos el planificador para transformar la máquina para que sea multi-aplicación, ya que de cara a la planificación, el que haya varias aplicaciones sólo se traduce en que se ha aumentado el número de hilos. Por lo tanto la planificación se hace igual.

El planificador de Jikes RVM utiliza un modelo de hilos M:N, esto quiere decir que la planificación de la ejecución de hilos de un número arbitrario (M) de hilos Java se realiza sobre un número finito (N) de pthreads. Esto significa que como máximo un número N de hilos Java se pueden ejecutar de forma concurrente.

N pthreads pueden crearse durante el tiempo de carga de la RVM, cada uno de los cuales comienza la ejecución de Java cambiando a una pila de un hilo Java y ejecutando el método run() de un hilo Java. Cuando un hilo Java se elimina de la planificación y otro hilo planificado lo reemplaza, el pthread salva el estado del hardware en ese momento en el hilo viejo, restaurando el estado del hardware del nuevo hilo cambiando de la vieja pila del hilo a la nueva pila.

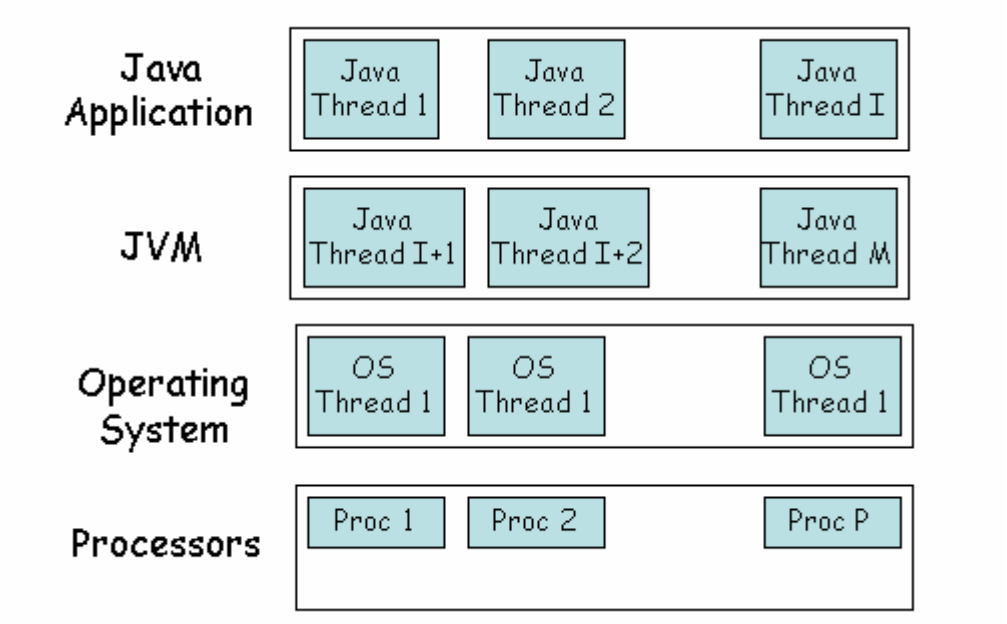


Figura 9-1: Distribución de hilos en Jikes RVM originalmente

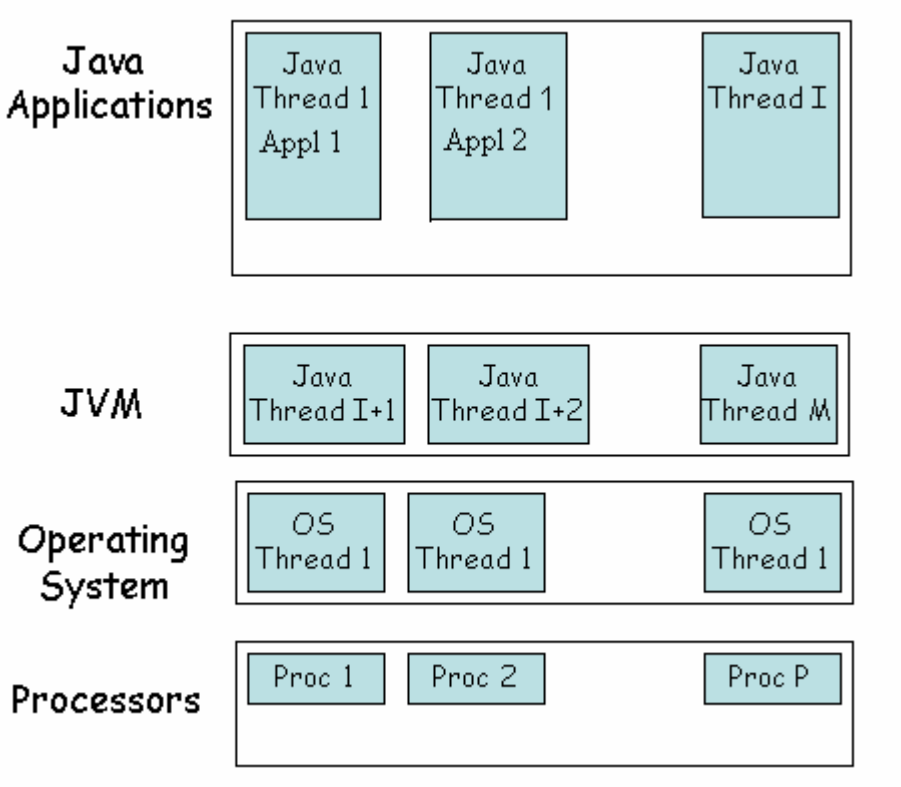


Figura 9-2: Distribución de los hilos según el esquema de nuestra máquina virtual

10 El gestor de memoria

10.1 Conceptos básicos del gestor de memoria

En este apartado vamos a tratar el gestor de memoria de Jikes RVM. Esta parte es la que más nos ha interesado, puesto que es la encargada de localizar en memoria los recursos que necesita la máquina virtual.

La herramienta JMTk (*Java Memory management Toolkit*) es el sistema de gestión de la memoria por defecto para Jikes RVM. JMTk ha sido diseñado para ser una herramienta portable.

Las clases *Plan* implementan los diferentes gestores de memoria y por lo tanto los diferentes recolectores de basura que están disponibles.

Todos los planes JMTk heredan de *BasePlan* y soportan localización y recolección paralela para evitar la sobrecarga que puede ser provocada por la sincronización. Las acciones asíncronas de los hilos locales se distinguen de las acciones globales que solamente pueden ser ejecutadas por un hilo simple.

Las funciones básicas de un plan incluyen:

- Identificar la distribución de la memoria virtual (esto se hace usando *VMResource* para, por ejemplo, asignar al *nursery* o al *ms* (*MarkSweep)* un rango determinado de direcciones). Es importante tener en cuenta que sólo la distribución de la memoria virtual se reparte de forma estática. El uso en un momento determinado de la memoria se extiende dinámicamente entre los diferentes espacios (meta-datos, inmortales, ms, nursery o los).

- Proporciona espacio para agrupar los diferentes asignadores de memoria necesarios para los diferentes *VMResources*.

- Invoca la recolección de basura cuando es necesario mediante el uso de un mecanismo de agrupamiento.

- Aplica las diferentes políticas de recolección de basura durante el proceso de recolección ya que los objetos pueden tener diferentes regímenes de recolección dependiendo del lugar en el que residan en memoria.

10.2 Tipos de recolectores de basura

Hay varios recolectores de basura, y por lo tanto gestores de memoria diferentes. En el momento de compilar y construir una versión ejecutable de la máquina se decide cual de ellos usar. A continuación mostraremos cuales son los que hay disponibles y sus características más significativas.

- SemiSpace. Este algoritmo utiliza dos espacios de copia del mismo tamaño. Las asignaciones se realizan de forma contigua en uno de los

espacios, y se reserva el otro para copiar los objetos por si se produce el peor de los casos, que es aquel en el que todos los objetos sobreviven.
Cuando está lleno, se copian los objetos vivos en el otro espacio y se intercambian. De esta forma, el tiempo de recolección es proporcional al número de objetos que sobreviven.
El rendimiento con este método de recolección se ve afectado porque se reserva la mitad del espacio para realizar las copias y los objetos que sobreviven durante un período de tiempo largo se copian repetidamente.

- MarkSweep. Un recolector de marcado y barrido que funciona en dos fases. Primero marca los bloques que son útiles y después realiza un barrido para eliminar los bloques que no han sido marcados en la fase anterior.

- GenCopy. Un recolector generacional de copiado que divide el espacio de memoria según la antigüedad de los datos. La recolección se realiza cuando el nursery (objetos *jóvenes*) está lleno, guardándose los supervivientes en un semi-espacio maduro.
  Cuando el semi-espacio maduro está lleno, se realiza la recolección en el heap entero.

- GenMS. Un recolector generacional, es decir que tiene en cuenta la antigüedad de los datos, con un espacio maduro de memoria que se gestiona a través de un mecanismo de marcado y barrido.

- CopyMS. Un recolector híbrido, lo que significa que combina la estrategia de copiado y la de marcado y borrado. Los nuevos objetos (*Nursery Heap*) se gestionan con un recolector de copia y los objetos más antiguos o supervivientes se colocan en el *MS Heap* y se gestionan con un recolector de marcado y barrido *(MarkSweep*).

- RefCount. Este recolector de basura utiliza una ubicación de Free-List y realiza recolecciones cíclicas y síncronas, es decir no concurrentes.
  En este caso el tiempo de recolección es proporcional al número de objetos borrados del heap.

- NoGC. En este caso no se realiza recolección de basura.

Ninguno de estos planes nos sirve sin realizar ninguna modificación para una máquina virtual multi-aplicación, puesto que no separan los espacios de memoria reservados a cada aplicación.

Lo que hemos hecho es partir de uno de ellos, en nuestro caso del plan CopyMS y modificarlo para independizar la memoria reservada a cada una de las aplicaciones y evitar que haya conflictos entre esos espacios de memoria.

10.3 Modificaciones en el gestor de memoria

En la máquina virtual original, la memoria se distribuye de la siguiente manera:

- **El heap para meta-datos.**
  Este heap se utiliza de forma auxiliar.

- **El heap para objetos largos.**
  En este heap se colocan los objetos de más de 2 KB.

- **El heap nursery.**
  En este heap se guardan los objetos más pequeños y en él se realiza una recolección de basura de copia.

- **El heap immortal.**
  Espacio que se utiliza para guardar las clases del recolector de basura y del gestor de memoria.

- **El MS heap.**
  En este heap se guardan los objetos más antiguos para los que se realiza una recolección de tipo marcado y barrido.

- **El heap boot.**
  Aquí se guarda la imagen de Jikes RVM generada en su proceso de construcción, con la que se inicia la ejecución de la máquina virtual.

Esta distribución se realiza al crear la imagen de Jikes RVM en la clase *Plan.java*, por lo tanto es en esta clase donde hemos realizado las modificaciones.

La clase *VMResource* es la clase que implementa los recursos de memoria virtual. Cada instancia de esta clase maneja una región contigua de memoria.

Sin embargo, de esta clase heredan otras tres clases diferentes que son *MonotoneVMResource*, *FreeListVMResource*, *ImmortalVMResource*. Estas clases son las que realmente se llaman para generar nuevos recursos de memoria.

La clase *MonotoneVMResource* implementa los recursos de memoria virtual monótonos.
Las instancias de esta clase responden a las peticiones de un espacio de direcciones virtual realizadas por el consumo continuo de recursos.

La clase *FreeListVMResource* implementa los recursos de memoria virtual de tipo *"free list".*
Las instancias de esta clase responden a las peticiones de un espacio de direcciones en memoria virtual realizadas por el consumo de recursos. Los consumidores pueden ser también recursos libres desde recursos posteriores.

La clase *ImmortalVMResource* hereda de la clase *MonotoneVMResource* y por lo tanto, mantiene sus restricciones. De esta forma se previene alguna posible eliminación de la memoria inmortal ya que es aquí donde está la funcionalidad que soporta la imagen de la máquina virtual y la eliminación de memoria accidentalmente en esta parte de la máquina provocaría un error grave.

Los espacios de memoria virtuales, *IMMORTAL_SPACE*, *BOOT_SPACE* y *METADATA_SPACE* se crean en la clase *BasePlan.java*, y que como hemos explicado anteriormente, en ellos se carga la imagen de la máquina virtual. Esta parte es común en todos los gestores de memoria y por ello esta clase no se encuentra aparte y de ella heredan los diferentes planes que gestionan la recolección de basura. Por esto, la hemos modificado.

Es en la clase *Plan.java* donde se crean los espacios de memoria virtual *NURSERY_SPACE*, *MS_SPACE* y *LOS_SPACE*.

Debido a que nuestra intención es ejecutar varias aplicaciones a la vez y que éstas se mantengan independientes unas de otras, hemos creado un heap de cada uno de los tipos indicados para cada una de las aplicaciones a ejecutar y otro diferente para la máquina virtual.

De tal manera que nos han quedado tres *NURSERY_SPACE*, tres *MS_SPACE* y tres *LOS_SPACE* tal y como mostramos en la figura 10-1.

```
// recursos de memoria virtual para la maquina virtual
private static MonotoneVMResource nurseryVM;
private static FreeListVMResource msVM;
private static FreeListVMResource losVM;


  //recursos de memoria virtual para la aplicacion uno
private static MonotoneVMResource nurseryVM1;
private static FreeListVMResource msVM1;
private static FreeListVMResource losVM1;


//recursos de memoria virtual para la aplicacion dos
private static MonotoneVMResource nurseryVM2;
private static FreeListVMResource msVM2;
private static FreeListVMResource losVM2;
```

Figura 10-1: Declaración de los recursos de memoria virtual para las distintas aplicaciones.

Pero la creación de estos espacios se realiza en memoria virtual por lo que tenemos que crearlos también en memoria física para que la independencia de las aplicaciones sea completa.

A continuación, en la figura 10-2, mostramos un ejemplo de la forma en que Jikes RVM realiza la reserva de memoria física y de la memoria virtual.

Para la implementación de memoria física, Jikes RVM utiliza la clase *MemoryResource*. Cada instancia de esta clase maneja un determinado número de páginas de memoria.

Estas operaciones se deben realizar para todos los recursos de memoria virtual que se declaren, de forma similar.

```
nurseryMR = new MemoryResource("nur", POLL_FREQUENCY);
msMR = new MemoryResource("ms", POLL_FREQUENCY);
losMR = new MemoryResource("los", POLL_FREQUENCY);
      nurseryVM = new MonotoneVMResource(NURSERY_SPACE,
      "Nursery", nurseryMR,   NURSERY_START, NURSERY_SIZE,
      VMResource.MOVABLE);
msVM = new FreeListVMResource(MS_SPACE, "MS", MS_START,
   MS_SIZE,VMResource.IN_VM);
losVM = new FreeListVMResource(LOS_SPACE, "LOS", LOS_START,
   LOS_SIZE, VMResource.IN_VM);
msSpace = new MarkSweepSpace(msVM, msMR);
losSpace = new TreadmillSpace(losVM, losMR);
```

Figura 10-2: Ejemplo de reserva de memoria fisica y de memoria virtual y su mapeo.

Finalmente, se declaran y se crean los *allocator* que utilizaremos para indicar donde se debe ubicar cada uno de los objetos que se vayan creando.

```
private BumpPointer nursery;
private MarkSweepLocal ms;
private TreadmillLocal los;

nursery = new BumpPointer(nurseryVM);
ms = new MarkSweepLocal(msSpace, this);
los = new TreadmillLocal(losSpace);
```

Figura 10-3. Creación de los allocator.

Las clases *BumpPointer*, *MarkSweepLocal* y *TreadmillLocal* heredan de *Allocator*. Estas clases actúan a modo de controladores sobre cada uno de los nuevos espacios creados en memoria.

10.4 Ubicación de objetos en memoria

10.4.1 Modificaciones en Plan.java para una máquina multi-aplicación.

Para calcular la ubicación que queremos que tenga un objeto en memoria la máquina virtual original utiliza el método *alloc()* de la clase *Plan.java* que se muestra en la figura 10-4.

Este método recibe como parámetros el tamaño que ocupa el objeto en bytes, un booleano para indicar si el objeto es un escalar y el tipo de ubicación del objeto y devuelve la dirección, como un tipo *VM_Address*, del primer byte de la región donde se va a localizar el objeto.

```
  public final VM_Address alloc(int bytes, boolean isScalar, int
allocator, AllocAdvice advice) throws VM_PragmaInline

{
    if (VM_Interface.VerifyAssertions)
VM_Interface._assert(bytes == (bytes & (~(BYTES_IN_ADDRESS-
1))));
    VM_Address region;
    if (allocator == NURSERY_SPACE && bytes >
LOS_SIZE_THRESHOLD) {
      region = los.alloc(isScalar, bytes);
```

```
    } else {
      switch (allocator) {
      case  NURSERY_SPACE: region = nursery.alloc(isScalar,
bytes); break;
      case       MS_SPACE: region = ms.alloc(isScalar, bytes,
false); break;
      case      LOS_SPACE: region = los.alloc(isScalar, bytes);
break;
      case IMMORTAL_SPACE: region = immortal.alloc(isScalar,
bytes); break;
      default:
        if (VM_Interface.VerifyAssertions)
          VM_Interface.sysFail("No such allocator");
        region = VM_Address.zero();
      }
    }
    if (VM_Interface.VerifyAssertions)
Memory.assertIsZeroed(region, bytes);
    return region;

}
```

Figura 10-4: Método alloc() de Plan.java.

Con el tipo *allocator* se indica el espacio de memoria donde se va a guardar el objeto, que como hemos explicado anteriormente puede ser *NURSERY_SPACE*, *MS_SPACE, LOS_SPACE* o *IMMORTAL_SPACE*.

En nuestra máquina virtual multi-aplicación hemos modificado este método, ya que para calcular la ubicación que debe tener un objeto además de tener en cuenta el tipo *allocator* del objeto debemos tener en cuenta a que aplicación pertenece el objeto que queremos guardar para mantener separados los objetos de una aplicación con respecto a las demás.

Para realizar esta distinción hemos utilizado un identificador de aplicación que tienen todos los hilos y que indican la aplicación a la que pertenece cada hilo.

De esta manera lo que hacemos es, en el momento de calcular donde debe ser alojado un objeto, es decir en el momento en que se llama a este método, comprobar cual es la aplicación que se está ejecutando (la máquina virtual, la aplicación 1 o la aplicación 2) y en función de este valor y sabiendo, de esta forma, a que aplicación pertenece cada objeto ubicar en el heap correspondiente el objeto.

Para saber cual de las aplicaciones se está ejecutando utilizamos el método *VM_Thread.getCurrentThread()* que te devuelve el hilo que se está ejecutando en ese momento y comprobamos cual es su identificador.

A continuación, en la figura 10-5 mostramos el código del método *alloc()* modificado.

```
public final VM_Address alloc(int bytes, boolean isScalar, int
allocator,
                                AllocAdvice advice)
    throws VM_PragmaInline {

    VM_Address region;
    if(VM_Thread.getCurrentThread().getIdAplic()==1) {
        if (allocator == NURSERY_SPACE && bytes >
LOS_SIZE_THRESHOLD) {
           region = los1.alloc(isScalar, bytes);
        }
        else
          switch (allocator) {
           case
NURSERY_SPACE:region=nursery1.alloc(isScalar,bytes);
break;
           case MS_SPACE: region = ms1.alloc(isScalar, bytes,
false); break;
           case LOS_SPACE: region = los1.alloc(isScalar, bytes);
break;
           case IMMORTAL_SPACE:region =
immortal.alloc(isScalar,bytes); break;
           default:
               region = VM_Address.zero();
            }
     }
     else{
     if(VM_Thread.getCurrentThread().getIdAplic()==2) {
           if (allocator == NURSERY_SPACE && bytes >
LOS_SIZE_THRESHOLD) {
                    region = los2.alloc(isScalar, bytes);
           }
        else {
           switch (allocator) {
           case NURSERY_SPACE: region =
nursery2.alloc(isScalar,bytes); break;
           case MS_SPACE: region = ms2.alloc(isScalar, bytes,
false); break;
           case      LOS_SPACE: region = los2.alloc(isScalar,
bytes); break;
           case IMMORTAL_SPACE: region =
immortal.alloc(isScalar, bytes); break;
           default:
               region = VM_Address.zero();
            }
       }
       }
      else{
         if (allocator == NURSERY_SPACE && bytes >
LOS_SIZE_THRESHOLD) {
            region = los.alloc(isScalar, bytes);
         } else {
      switch (allocator) {
      case  NURSERY_SPACE: region = nursery.alloc(isScalar,
bytes); break;
      case       MS_SPACE: region = ms.alloc(isScalar, bytes,
false); break;
      case      LOS_SPACE: region = los.alloc(isScalar, bytes);
break;
      case IMMORTAL_SPACE: region = immortal.alloc(isScalar,
bytes); break;
      default:
        region = VM_Address.zero();
```

```
      }
       }
  }
}
return region;
}
```

Figura 10-5: Método alloc() de nuestro plan.java.

Debido a que en el *heap immortal* se guarda la imagen de la máquina virtual no hemos duplicado este espacio de memoria para cada una de las aplicaciones de tal manera que sólo hay un *IMMORTAL_SPACE*.

10.4.2 Traza detallada de la asignación de memoria para un objeto

Una pregunta que nos hicimos durante el desarrollo del proyecto y que nos llevó bastante tiempo resolver fue como se ubicaban en memoria las instancias de las clases que se creaban en nuestras aplicaciones (con la instrucción **new**).

Finalmente averiguamos que cuando la máquina se encuentra con una instrucción de este tipo se invocaba a los métodos de ubicación de objetos que están en la clase *RunTime*.

Pasaremos a mostrar cual es la secuencia de invocaciones de métodos que se realiza y a explicar como funcionan algunos de estos métodos.

El primer método que se llama es el *Object unresolvedNewScalar(int id)*. El parámetro que se pasa como entrada indica la referencia que especifica el tipo de la clase del nuevo objeto que queremos crear y devuelve dicho objeto con la cabecera incluida (ver el apartado referente a las estructuras de datos) y con sus campos inicializados a cero.

Para poder realizar esta operación, este método llama al *resolvedNewScalar()* que devuelve el objeto, pero para que este método pueda realizar su función se deben llevar a cabo unos pasos que detallamos a continuación.

- Primero tenemos que transformar el tipo que se pasa como parámetro en una clase y conociendo la clase podemos ir obteniendo el resto de datos que necesitamos.

- Necesitamos conocer el *allocator* de la clase. Este valor nos da la información, en función de la clase, del lugar de memoria donde debe ubicarse la instancia de la clase, que puede ser *IMMORTAL_SPACE*, *MS_SPACE* o *NURSERY_SPACE*.

- También necesitamos conocer el desplazamiento (*offset*) de la clase, es decir el tamaño que van a ocupar las instancias de esta clase en memoria. Como es natural, este valor depende del número de atributos que tiene la clase.

El cálculo del valor *allocator* se realiza haciendo una llamada al método *MM_Interface.pickAllocator().* Consideramos que el funcionamiento de este

método es muy interesante por lo que mostramos su código en la figura 10-6 y vamos a pasar a explicar su funcionamiento de una forma más detallada.

```
public static int pickAllocator(VM_Type type, VM_Method method)
    throws VM_PragmaInterruptible {

    if (method != null) {
     // We should strive to be allocation-free here.
      VM_Class cls = method.getDeclaringClass();
      byte[] clsBA = cls.getDescriptor().toByteArray();
      if (VM_Interface.GCSPY) {
        if (isPrefix("Lorg/mmtk/vm/gcspy/",  clsBA) ||
            isPrefix("[Lorg/mmtk/vm/gcspy/", clsBA)) {
       return Plan.GCSPY_SPACE;
        }
      }
      if (isPrefix("Lorg/mmtk/", clsBA)
       ||
isPrefix("Lcom/ibm/JikesRVM/memoryManagers/mmInterface/VM_GCMapI
teratorGroup", clsBA)) {
        return Plan.IMMORTAL_SPACE;
      }
    }
    MMType t = (MMType) type.getMMType();
    return t.getAllocator();
  }
```

Figura 10-6: El método pickAllocator(VM_Type).

Este método devuelve el lugar en memoria donde deben ser ubicados los objetos de una clase.
Para realizar esta distinción lo primero que tiene en cuenta es el paquete en el que está una clase, es decir el prefijo de la misma.

Si el prefijo de la clase es *org/mmtk/vm/gcspy/* significa que la clase pertenece al recolector de basura, en este caso las instancias de esta clase se deben ubicar en el *GCSPY_SPACE*, que en nuestro caso coincide con el *IMMORTAL_SPACE*.

Las clases que pertenecen al paquete *org/mmtk,* es decir, las clases que implementan el gestor de memoria, se deben guardan en el *IMMORTAL_SPACE*.

Para el resto de las clases se mira cual es su *MMType*, y en función de su valor se decide cual debe ser su lugar de ubicación en memoria. Este valor es un atributo de la clase *VM_Type* y se calcula cuando se resuelve la clase para cargarla en el JTOC, es decir cuando se invoca el método *resolve()* de la clase.

Para ello se utiliza el método *MM_Interface.notifyClassResolved()* que cuando un nuevo tipo ha sido resuelto por la máquina virtual, crea un nuevo tipo *MM* (memoria física) para reflejar el tipo *VM* (memoria virtual) y asocia el tipo *MM* con el tipo *VM*.

De este modo, las instancias de la clase *MM_type* encapsulan la información que especifica el tipo, que utilizará el gestor de memoria para decidir donde ubicar las instancias de los objetos.

Con estos valores conocidos podemos llamar al método *resolvedNewScalar()* cuyo funcionamiento detallamos a continuación.

Con la ejecución del método *resolvedNewScalar*() de la clase *VM_RunTime.java*, una instancia de una clase es ubicada invocando el método *allocateScalar()* de la clase *MM_Interface.java.*

El método *allocateScalar()* devuelve un objeto totalmente definido e inicializado de la clase correspondiente, que a su vez llama al método *plan.alloc(),* que hemos explicado y modificado anteriormente.

10.5 Modificaciones en la recolección de basura

La recolección de basura tiene en cuenta el lugar donde está guardado un objeto, por lo tanto al haber modificado la distribución de la memoria y la ubicación de objetos, nos hemos visto obligados a realizar algunas modificaciones en la forma de realizar la recolección aunque siempre manteniendo los fundamentos en los que se basa el recolector CopyMS. Puesto que es de este colector del que hemos partido para realizar el proyecto.

Así pues, además de modificar el método *alloc()* de la clase *Plan.java* también hemos tenido que modificar otros métodos asociados a éste.

A continuación pasaremos a comentar algunas de estas modificaciones:

- El método *allocCopy()* que busca espacio en memoria para copiar un objeto necesita tener en cuenta donde estaba guardado inicialmente el objeto para, dependiendo de esto, guardarlo en el *MS_SPACE* correspondiente a la aplicación que se está ejecutando.

- El método *pool()* se llama periódicamente por el subsistema de ubicación (por defecto cada vez que una página es consumida), y da al recolector la posibilidad de realizar la recolección de basura.
  La máquina original por tener sólo un *heap nursery* comprueba si está lleno y si es así fuerza la recolección, sin embargo en nuestro caso al tener varios *heaps nursery* lo que hacemos es forzar la recolección cuando algunos de ellos está lleno, de tal modo que los comprobamos todos.

- También hemos modificado los métodos que se utilizan para hacer las trazas del consumo de memoria, de tal manera que se tengan en cuenta los nuevos heaps que hemos creado. Añadiendo sus páginas de memoria al consumo total y mostrando también los nuevos heaps.

11 Benchmarks

Para probar las modificaciones que hemos realizado en la máquina virtual y hacer una comparativa con la versión de Jikes RVM original, hemos utilizado seis benchmarks SPECjvm98.

- Mpegaudio descomprime ficheros de audio de acuerdo con las especificaciones de ISO MPEG Layer-3. La carga de trabajo se encuentra alrededor de los 4MB de datos de audio.
  Recolección de basura. El uso del recolector de basura es insignificante.
  Desarrollado por el Fraunhofer Institut fuer Integrierte Schaltungen.

- Mtrt es la versión del 205.raytrace. Trabaja en una escena gráfica de un dinosario. Dos hilos realizan el renderizado de una escena.
  Recolección de basura. Se ejecuta con 16MB de heap y almacena 355MB de objetos.
  Desarrollado por Sun Microsystems.

- Javac es el compilador de Java para el JDK 1.0.2. Es una aplicación comercial por lo que su código fuente no se proporciona.
  Recolección de basura. Se ejecuta con 12MB de heap y almacena 518MB de objetos.
  Desarrollado por Sun Microsystems.

- Jack es un parser de Java basado en el Purdue Compiler Construction Tool Set (PCCTS). Un parser es un analizador sintáctico que analiza una cadena de símbolos. Su carga de trabajo consiste en un archivo llamado jack.jack, que contiene las instrucciones para que jack se genere a sí mismo.
  Recolección de basura. Este programa se ejecuta con 2MB de heap y almacena 481MB de objetos.
  Lo desarrolló Sun Microsystems.

- Jess es la versión Java del sistema experto CLIPS de la NASA. Está compuesto fundamentalmente de estructuras if-then y llamadas a reglas almacenadas como un conjunto de datos.
  Recolección de basura. Este programa se ejecuta con 2MB de heap y almacena 748MB de objetos.
  Ha sido desarrollado en los Sandia National Laboratories.

- Raytrace realiza un renderizado de una imagen 3D creada con gráficos por computadora.

12 Resultados de las pruebas

Antes de pasar a ver los resultados de las pruebas es necesario tener en cuenta las condiciones bajo las que se han llevado a cabo. Para empezar, las pruebas han sido realizadas en un nodo perteneciente a un cluster de DACYA (Departamento de Arquitectura de Computadores y Automática) de la Universidad Complutense de Madrid, el cual estaba sometido a cargas de trabajo difíciles de determinar.

Debido a esto, las medidas de tiempo no se deben tomar como tiempo absoluto de ejecución, sino como una medida para comparar la ganancia de tiempo de ejecución. Las pruebas fueron realizadas en serie un mismo día y separadas únicamente por los segundos necesarios para preparar la ejecución de la siguiente prueba buscando minimizar el impacto que la variación de carga del cluster pudiese tener en el resultado final.

Otro aspecto a tener en cuenta es que, debido a las estrategias que hemos empleado para dividir la memoria reservada por Jikes RVM entre las distintas aplicaciones, nos encontramos con que la máquina original dispone de aproximadamente el doble de espacio reservado para una aplicación en comparación con nuestra máquina, aunque en la máquina original dicho espacio es compartido por la aplicación y por la propia máquina. Esta disminución del espacio disponible para una aplicación dada aumenta la probabilidad de que sea necesario realizar una recolección de basura.

Por último hay que resaltar que en nuestra máquina modificada, al lanzar el recolector de basura (porque alguno de los heaps se haya llenado) dicha recolección se realiza sobre todos los espacios, liberando de basura tanto el espacio de la máquina como el de todas las aplicaciones.

Las pruebas se han realizado comparando ambas máquinas tanto a la hora de lanzar una única aplicación como lanzando dos simultáneamente. El tamaño de datos empleado ha sido uno.

12.1 Resultados del consumo de memoria

Estos son los resultados del consumo de memoria en las distintas pruebas (consultar ANEXO D):

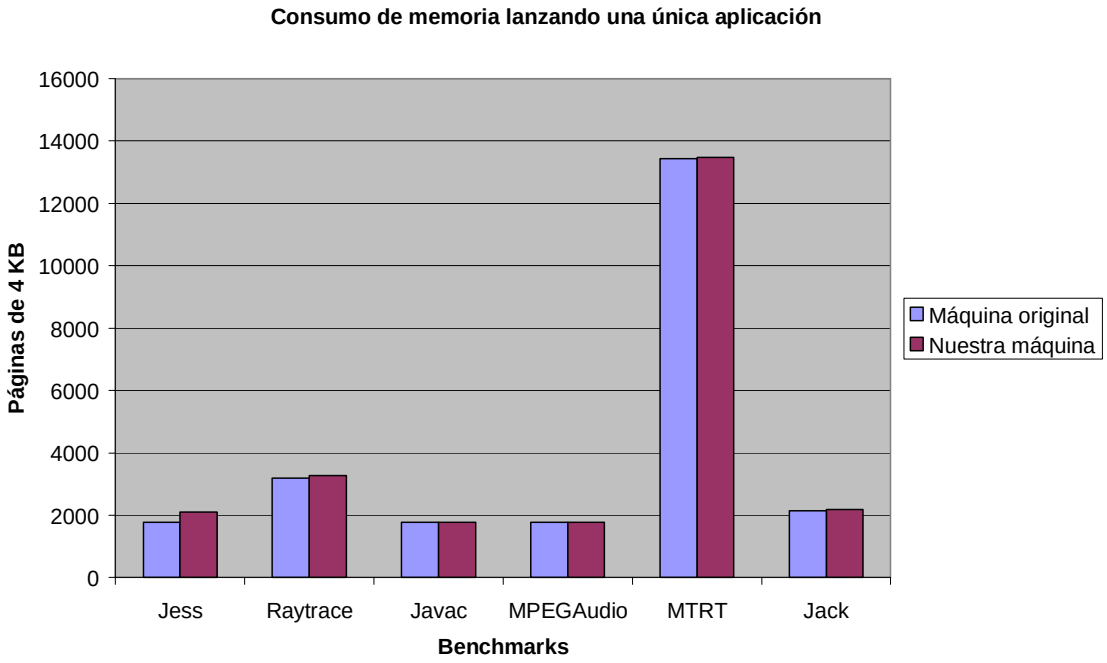


Figura 12-1: Gráfica del consumo de memoria lanzando una única aplicación.

A la hora de ejecutar una única aplicación no hay diferencias apreciables pues para este caso ambas máquinas se comportan de la misma manera, aunque nuestra máquina reserva la mitad de memoria para la aplicación.

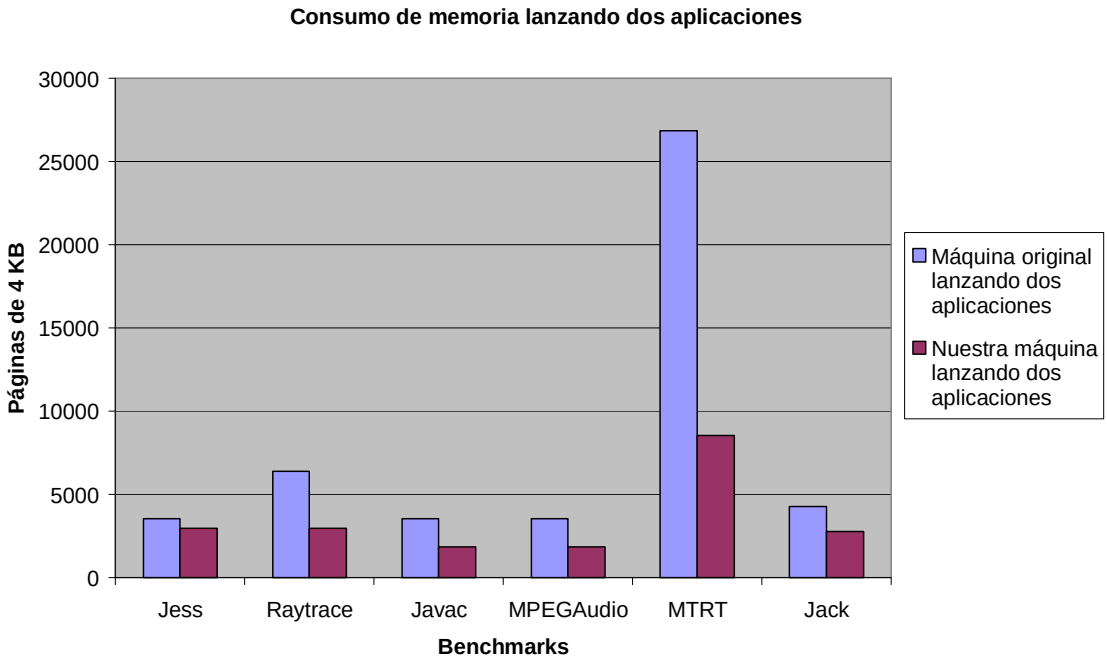


Figura 12-2: Consumo de memoria lanzando dos aplicaciones.

Es a la hora de ejecutar dos aplicaciones cuando podemos ver mejoras significativas. Para empezar, la máquina original debe reservar el doble de espacio debido a que se deben lanzar dos imágenes de la máquina para poder ejecutar dos aplicaciones simultáneamente. Nuestra máquina modificada divide el espacio reservado originalmente para la máquina y una única aplicación en tres partes iguales: una para la máquina y una para cada aplicación.

La mejora en memoria no se debe únicamente a la reducción del espacio del que dispone una aplicación. Recordemos que gracias al sistema de carga dinámica de clases de Jikes RVM, si una aplicación necesita cargar una clase que ya ha sido previamente cargada por la otra aplicación o por la máquina, no será necesario volver a cargarla. Gracias a esto si las aplicaciones utilizan paquetes o librerías en común, éstas se cargarán una única vez.

Como ya hemos dicho, el menor espacio disponible por aplicación conlleva un posible aumento del número de recolecciones de basura necesarias (más adelante trataremos este aspecto en detalle). Pero gracias a que cada recolección libera memoria en todos los espacios estamos evitando el riesgo de que se produzca una sucesión de recolecciones.

12.2 Resultados del tiempo de ejecución.

A la hora de medir el tiempo de ejecución lanzando dos aplicaciones, para la máquina original simplemente hemos multiplicado por dos los valores obtenidos al lanzar una aplicación (consultar ANEXO D):

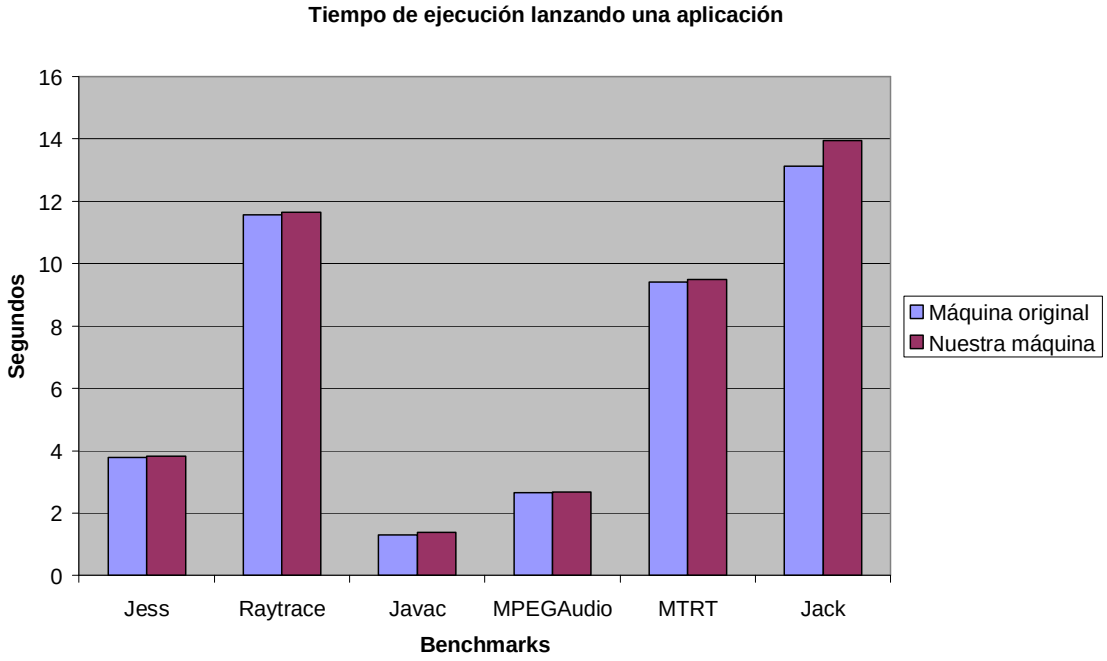


Figura 12-3: Tiempo de ejecución lanzando una aplicación.

En la gráfica anterior se puede ver que la máquina original que utiliza para una aplicación el doble de memoria que la nuestra, es igual de rápida.

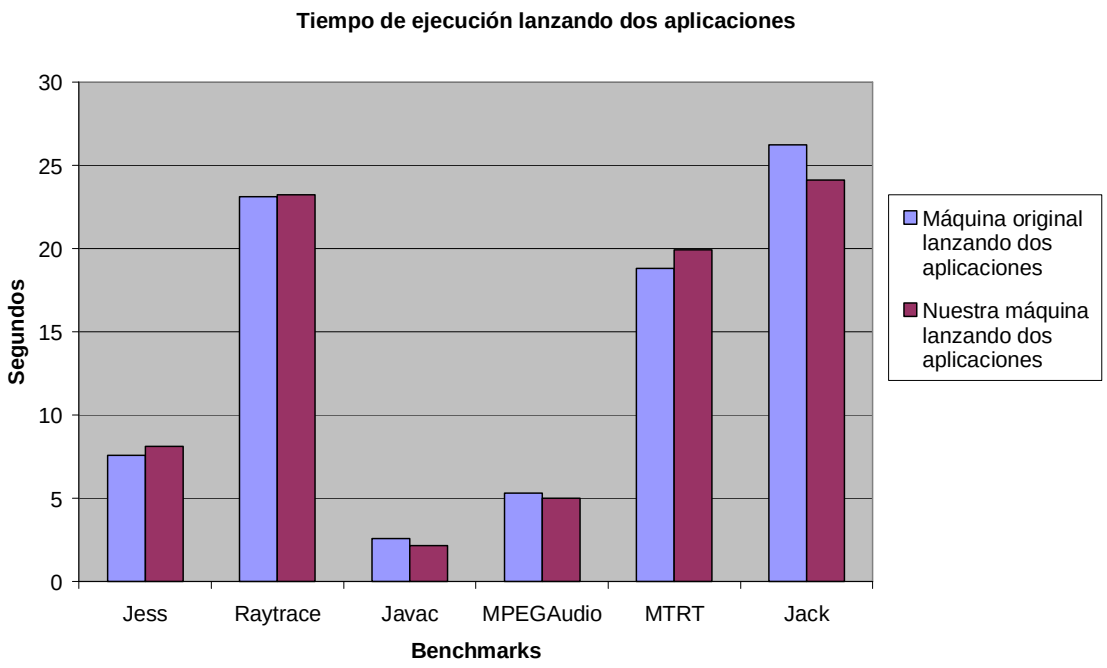


Figura 12-4: Tiempo de ejecución lanzando dos aplicaciones.

En el caso de ejecutar dos aplicaciones, se puede observar en la gráfica como la pérdida de tiempo debida al mayor número de recolecciones ha sido compensada con la ganancia provocada al cargar una única imagen de la máquina y no repetir cargas de clases comunes.

Efectivamente nuestra máquina se ha visto perjudicada por el mayor número de recolecciones de basura que ha tenido que realizar. No obstante, esta pérdida se ve compensada con una ganancia en tiempo debida a que se invierte menos tiempo en cargar tanto la propia máquina virtual (sólo se carga una vez) como en cargar clases compartidas por las aplicaciones. Gracias a esto el tiempo de ejecución se mantiene similar al de la máquina original.

12.3 Número de recolecciones de basura.

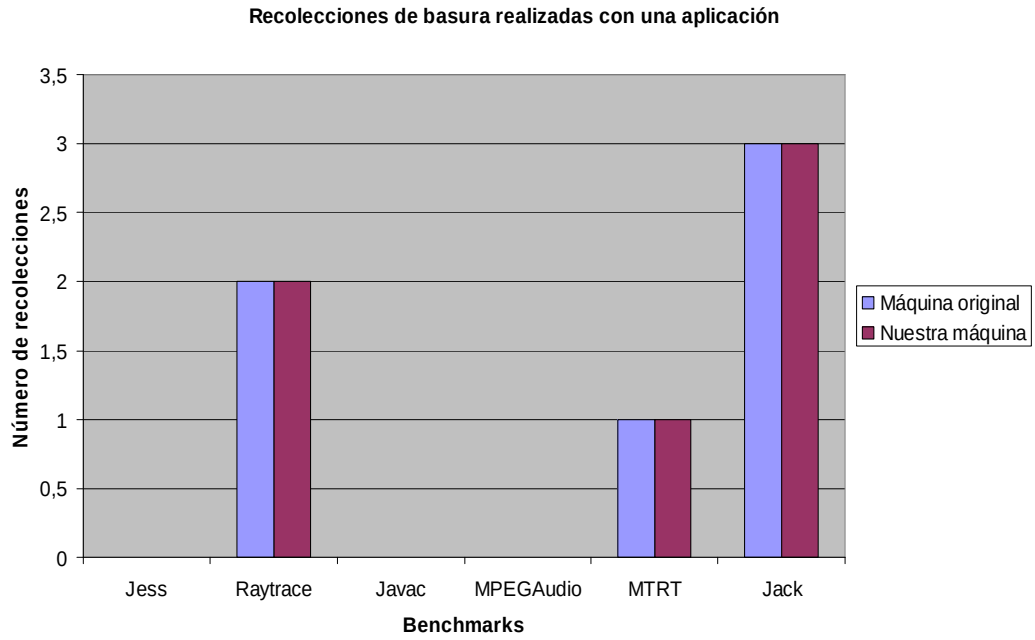


Figura 12-5: Recolecciones de basura lanzando una única aplicación.

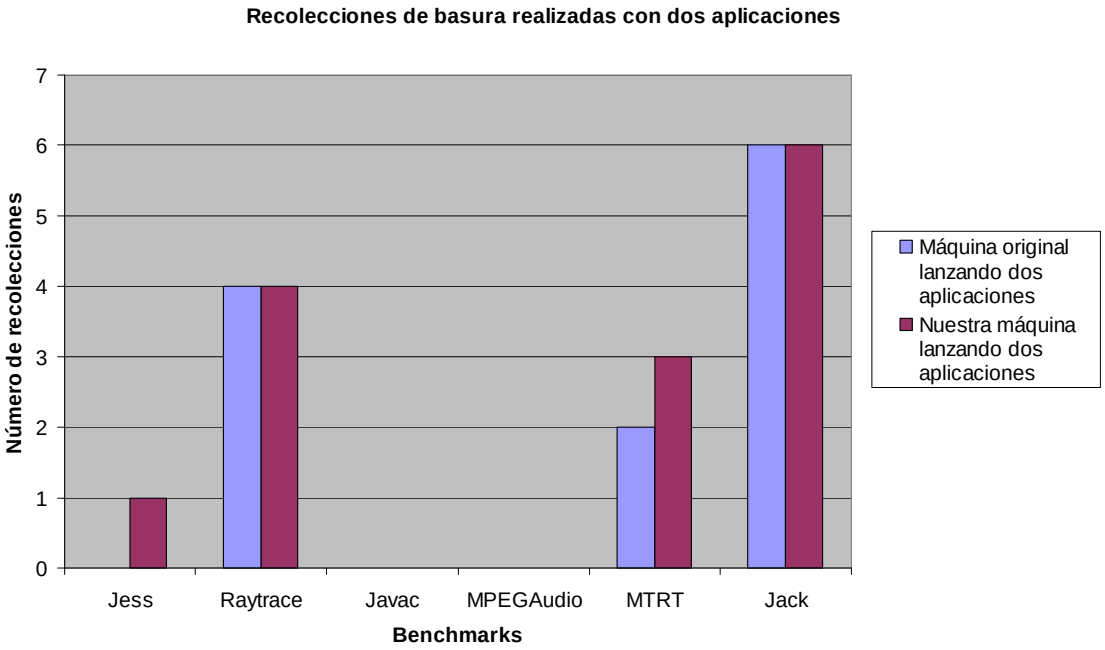


Figura 12-6: Recolecciones de basura lanzando dos aplicaciones

Si bien sospechábamos que un menor espacio disponible para las aplicaciones iba a suponer una mayor frecuencia de recolecciones, las pruebas indican que este efecto no es tan acusado como cabría esperar.

De hecho, el número de recolecciones al lanzar una sola aplicación ha sido el mismo en ambas máquinas (ver ANEXO D).

En cuanto al lanzamiento de dos aplicaciones en dos casos realizamos una recolección de más. Esto se debe a que los heaps son más pequeños.

Se podría esperar un número mucho mayor de recolecciones en nuestra máquina respecto de la original. Sin embargo, reducimos el número de recolecciones porque se libera memoria de todos los espacios, no sólo del que ha provocado la recolección lo que disminuye la probabilidad de que se lance de nuevo el recolector de basura.

13 Conclusiones

Hemos obtenido una máquina virtual que ejecuta dos aplicaciones en el mismo espacio de memoria en el que la máquina original sólo ejecuta una y empleando únicamente una imagen de dicha máquina, sin que por eso se vea afectado su rendimiento en tiempo de ejecución.

En la tabla siguiente podemos apreciar los porcentajes de mejora del rendimiento tanto en memoria como en tiempo.

Las variaciones están calculadas con respecto a la máquina original.

		Variación de memoria	Variación de tiempo
Con una aplicación	Jess	+18,5%	+0,94%
	Raytrace	+2,5%	+0,72%
	Javac	+0%	+6,8%
	MPEGAudio	+0%	+0,79%
	MTRT	+0,4%	+0,86%
	Jack	+2,2%	+6,3%
	Promedio	+3,9333%	+2,57%
Con dos aplicaciones	Jess	-16,8%	+6,8%
	Raytrace	-53%	+0,54%
	Javac	-48%	-16,6%
	MPEGAudio	-48%	-5,8%
	MTRT	-68,1%	+5,4%
	Jack	-35,8%	-8%
	Promedio	-44,95%	-2,94%

Figura 12-1: Ganancias.

Aunque en un principio pudiera parecer que el que estén almacenados en una misma estructura (JTOC) todos los miembros estáticos y la información de las diferentes clases de las distintas aplicaciones puede afectar seriamente a la seguridad de nuestro sistema, la independencia de las aplicaciones está garantizada gracias al compilador (y al lenguaje de programación Java): ninguna aplicación hará uso de atributos o métodos estáticos de ninguna clase que no haya declarado, o importado, previamente.

También el hilo principal de una aplicación y su pila asociada se guardan en el espacio de memoria reservado para la máquina virtual. De nuevo, como los objetos pila son privados a cada hilo, el lenguaje Java nos asegura la no interferencia. Tampoco se producen sobreescrituras y borrados accidentales de los hilos de una aplicación por parte de otra ya que la propia máquina ofrece mecanismos para controlar la correcta asignación de memoria.

Se podría, en un futuro, realizar una separación de las pilas de los hilos principales y ubicarlas en el espacio dedicado a cada aplicación.

Ha resultado un acierto el permitir que el recolector de basura libere memoria en todos los espacios en lugar de restringir su acción únicamente al

espacio cuya saturación provoca la recolección. De esta forma, no se incrementa en exceso el número de recolecciones.

No obstante, en el caso de un número elevado de aplicaciones simultáneas, es posible que el rendimiento en tiempo de ejecución sufra una degradación importante. El recolector de basuras de Jikes RVM provoca que el resto de hilos activos tengan que esperar a que finalice la recolección. En el caso de tener una aplicación intensiva en consumo de memoria se mantendrían todas las aplicaciones bloqueadas en espera de una sola de ellas.

Una posible mejora podría ser la modificación del recolector de basura para que únicamente recorriese el espacio agotado, bloqueando nada más la aplicación responsable, permitiendo a los hilos del resto de aplicaciones continuar con su ejecución normal. Con los mecanismos de seguridad adecuados, estos hilos activos no deberían interferir con la recolección.

Otra mejora sería estudiar la cantidad de memoria que se concede a cada aplicación para disminuir el tiempo y el número de recolecciones de basura.

ANEXO A. Introducción a las máquinas virtuales. Máquina virtual de Java.

Una máquina virtual es un software que crea un entorno sobre un hardware emulando una máquina física. De esta manera conseguimos una separación entre la máquina real y el software que se esté ejecutando, ya sea una aplicación cualquiera o incluso un sistema operativo completo. Nosotros nos vamos a centrar en el primer caso, una máquina virtual destinada a la ejecución de aplicaciones; concretamente la máquina virtual de Java.

Java es un lenguaje de programación orientado a objetos creado por *Sun Microsystems* en 1995. Uno de los objetivos principales de este lenguaje se puede resumir con la frase *"compile once, execute many"*. Se buscaba el evitar que fuese necesario que para cada plataforma hardware hubiese que realizar una compilación del código fuente. Este objetivo va más allá de la portabilidad de código. La motivación por la que *Sun Microsystems* se plantea este objetivo es la proliferación de Internet como una red completamente heterogénea de ordenadores interconectados.

La solución a este problema se basa en realizar una compilación en dos fases. La primera fase se corresponde con la compilación normal de cualquier lenguaje de alto nivel, solo que el código objeto obtenido, en lugar de ser específico para una plataforma hardware dada es un lenguaje intermedio denominado *bytecode*. Dicho lenguaje intermedio es posteriormente compilado o interpretado para un hardware concreto. Esta segunda fase es la que realiza la Máquina Virtual de Java.

Otra gran ventaja del lenguaje de programación de Java, además de su portabilidad, es la seguridad. Como tienes una máquina virtual que ejecuta el código de las aplicaciones, cualquier tipo de problema que se produzca en el transcurso de de la ejecución no afectará a la máquina física.

La propia máquina virtual comprueba que en código byte no se acceda a direcciones de memoria restringidas. También permite ejecución segura de código remoto.

Esta máquina virtual desarrollada por *Sun Microsystems* hace de intermediaria entre la aplicación Java (en *bytecode*) y el hardware de la máquina física. Pero la funcionalidad de esta máquina virtual va más allá de compilar/interpretar el código byte a código máquina. A continuación exponemos las funciones que realiza la máquina virtual de java.

- Abstracción de una arquitectura
  - portabilidad de Java.
  - Interfaz entre la aplicación y el sistema operativo y el hardware.

- Proveedor de servicios para las aplicaciones Java
  - Compilación (traducción/ interpretación) de los *Bytecodes*.
  - Gestión de los hilos de ejecución.

- Sincronización
- Conmutación entre hilos.

o Tareas en tiempo de ejecución.
- Carga dinámica de clases.
- Control de tipos dinámico.
- Gestión de las excepciones.
- Gestión de las interfaces.
- Entrada/ salida.
- Llamada a métodos nativos.
- Reflexividad.

o Gestión de la memoria
- Asignación.
- Recolección de basura.

Vemos pues que la Máquina Virtual de Java ofrece la abstracción de una arquitectura junto con servicios propios de un sistema operativo. Gracias a esto, cualquier desarrollo Java se hará con el mismo modelo hardware en mente, con las ventajas que esto conlleva, ya que será la Máquina Virtual la que permita ejecutar la aplicación en una plataforma concreta.

Una vez expuesto este planteamiento teórico, junto con lo comentado anteriormente de la creciente importancia de Internet, podemos ver tanto el resultado obtenido como la gran repercusión que ha tenido. Con esta situación, una persona puede crear una aplicación escrita en Java, compilarla una única vez y ponerla disponible en Internet. No importará la máquina que se utilice; cualquier persona podrá descargarse dicha aplicación y ejecutarla en su ordenador directamente, sin necesidad de realizar una compilación previa.

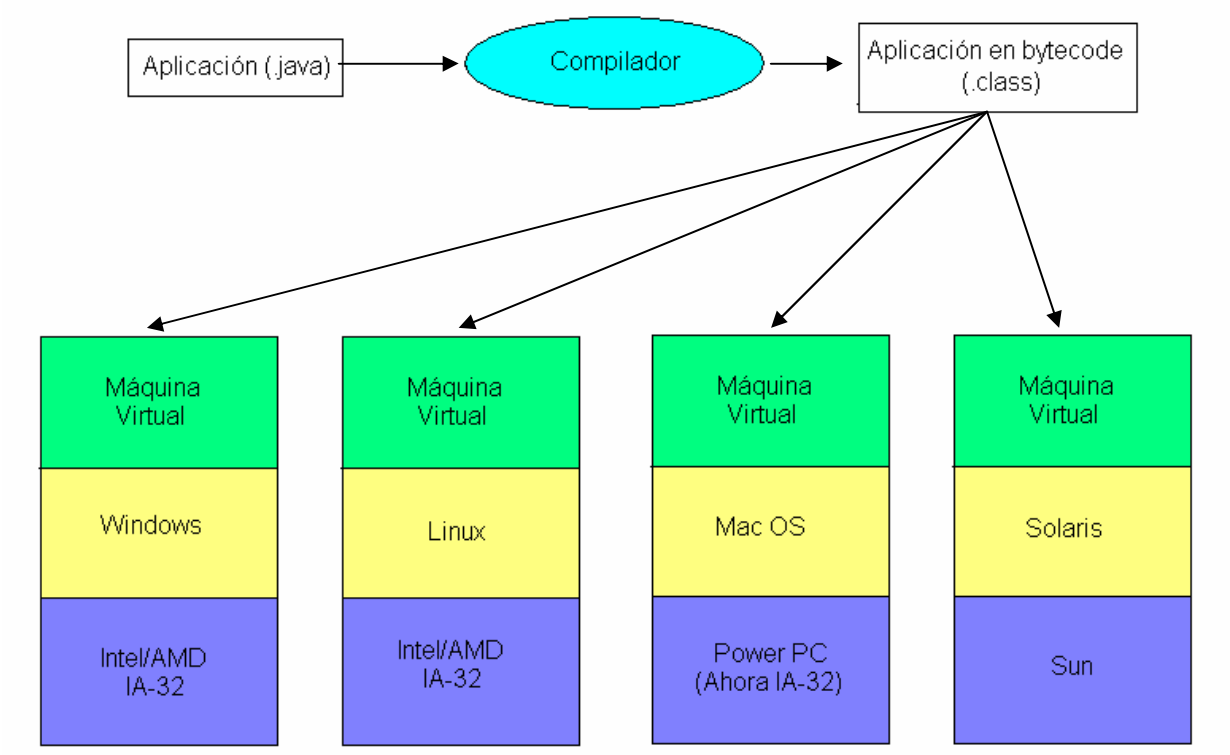


Figura A-1: Filosofía de Java y su máquina virtual.

Lógicamente será necesario crear una Máquina Virtual de Java para cada plataforma hardware (Intel IA-32, PowerPC…) y sistema operativo (Windows, Linux, Mac OS, Solaris…).

Podemos ver el gran éxito del lenguaje Java y su filosofía de máquina virtual en una gran cantidad de hechos:

- La cada vez mayor variedad de dispositivos que se conectan a Internet, como las PDA´s o los teléfonos móviles, que incorporan la posibilidad de ejecutar aplicaciones Java (en este caso J2ME: Java 2 Mobile Edition).

- El impulso definitivo que ha propiciado que multitud de empresas salten de aplicaciones antiguas escritas en lenguajes como Cobol a aplicaciones basadas en Web (J2EE).

- El gran uso de tecnologías Java para crear contenidos en la Web (JavaScript, JSP, AJAX…).

Incluso se ha aplicado este concepto extrapolándolo a los futuros reproductores de Blu-ray Disc, los cuales ejecutarán código Java para crear, por ejemplo, mejores menús en las películas. En este caso, en lugar de publicar la aplicación en Internet se publica en los discos Blu-ray, mientras que la Máquina Virtual de Java permite su ejecución en cualquier reproductor de Blu-ray Disc.

En el lado negativo de esta filosofía nos encontramos con la necesidad de tener un software ejecutándose en nuestra máquina además de la aplicación que nos interesa, junto con una compilación/interpretación de esta última “al vuelo”, con las penalizaciones de velocidad de ejecución y memoria que todo esto plantea. No obstante se están haciendo notables progresos en su solventación.

ANEXO B: Instalación de Jikes

Para realizar el proyecto hemos partido de la máquina virtual de Java Jikes RVM. La versión que hemos instalado es Jikes RVM 2.3.2.

Para realizar la instalación de la máquina virtual adecuadamente hemos seguido los siguientes pasos:

1. Lo primero fue instalar una máquina virtual de Java diferente a Jikes RVM, en nuestro caso elegimos la j2sdk 1.4.1 que descargamos de la página Web *http: //www.blackdown.org/.* La instalación de esta máquina virtual es necesaria para que Jikes RVM pueda compilarse inicialmente, ya que al ser código Java utiliza otra máquina virtual diferente a ella misma para cargarse.

2. Instalar el compilador de Jikes. La versión que hemos descargado nosotros es Jikes Compiler 1.18 de la página Web *http: //jikes.sourceforge.net/*

3. Descargarse Jikes RVM 2.3.2. de la página Web *http: //jikesrvm.sourceforge.net/*, y descomprimirla. La descarga de la máquina virtual la hemos hecho en un directorio de trabajo creado para este propósito y que hemos llamado *rvmRoot* donde hemos guardado los archivos fuente de la máquina virtual, las librerías fuente y las herramientas necesarias para la construcción de Jikes RVM.

4. También hemos necesitado la versión 0.08 de las librerías GNU Classpath. La hemos obtenido en la página Web *http: //gnu.org/software/classpath.*

5. Para que Jikes RVM funcione correctamente fue necesario crear unas variables de entorno en Linux requeridas para el correcto funcionamiento de la máquina virtual. Estas son las que hemos definido:

   RVM_ROOT
   : Es la localización de la distribución de Jikes RVM. En nuestro caso esta variable apunta a $HOME/rvmRoot

   RVM_HOST_CONFIG
   : Es el archivo de configuración de la máquina en la que se compila Jikes RVM.
   En nuestro caso y debido a la arquitectura y sistema operativo que utilizamos el valor de esta variable es: $RVM_ROOT/rvm/config/i686-pc-linux-gnu

   RVM_TARGET_CONFIG
   : Es la configuración de la máquina que ejecuta Jikes RVM. Este será normalmente $RVM_ROOT/rvm/config/i686-pc-linux-gnu que coincide con el valor de RVM_HOST_CONFIG, de hecho si no se

le asigna ningún valor a esta variable de entorno el programa de configuración *jconfigure* asumirá que su valor es el mismo que el de la variable de entorno citada anteriormente.

RVM_BUILD

Es el directorio donde el proceso de compilación y construcción va a generar una configuración ejecutable de Jikes RVM. En nuestro caso esta variable de entorno apunta al directorio $HOME/rvmBuild.

PATH

Nuestro path debe contener $RVM_ROOT/rvm/bin para la correcta ejecución de varios scripts y utilidades.

Para la definición de estas variables de entorno hemos modificado el archivo de configuración *.bashrc*. Las líneas introducidas han sido las siguientes:

```
RVM_ROOT=$HOME/rvmRoot
export RVM_ROOT

RVM_BUILD=$HOME/rvmBuild
export RVM_BUILD

RVM_HOST_CONFIG=$RVM_ROOT/rvm/config/i686-pc-linux-gnu.kaffe
export RVM_HOST_CONFIG

RVM_TARGET_CONFIG=$RVM_ROOT/rvm/config/i686-pc-linux-gnu.kaffe
export RVM_HOST_CONFIG

#La siguiente definicion del PATH seria conveniente mejorarla
con un
# "if" para evitar definir cosas más de una vez

PATH=$PATH:$RVM_ROOT
PATH=$PATH:$RVM_BUILD
```

Figura B-1: Declaración de las variables de entorno.

El archivo *.bashrc* se carga cada vez que se abre sesión mediante un terminal. Puede parecer más adecuado realizar esta modificación en el archivo *.bash_profile*, el cual se carga con cada sesión abierta mediante entorno de ventanas. No obstante, resulta más adecuada nuestra decisión principalmente por dos motivos: por un lado la mayoría de las operaciones con Jikes RVM se hacen por consola, y por otro podemos abrir sesiones por SSH para trabajar remotamente.

6. Editar un archivo de configuración para que la máquina virtual encaje con la máquina donde va a instalarse. Se debe editar un script en el directorio $RVM_ROOT/rvm/config para montar las variables que se usarán en el proceso de instalación.

Estos archivos de configuración tienen dos secciones. En la primera sección, se debe especificar el sistema operativo y la arquitectura.

En nuestro caso, para el sistema operativo definimos RVM_FOR_LINUX, para la arquitectura definimos RVM_FOR_IA32 .

La segunda sección del archivo de configuración se utiliza para definir como encontrar las herramientas que Jikes RVM necesita. Se deben poner las siguientes variables:

HOST_JAVA_HOME
: Tiene de definir el directorio base del JDK JVM, necesario para la construcción de la máquina virtual.

CLASSPATH_ROOT
: El directorio padre del que contiene el código fuente del paquete de clases GNU y se ha descargado manualmente, que es lo que hemos hecho nosotros. Este paso suele ser crítico.

7. Montar un directorio para la construcción de la máquina virtual Jikes RVM con el comando *jconfigure <boot image compiler>{"Base"/"Adaptive"} <garbage collector>.* En primer parámetro indica el compilador que se va a utilizar para compilar y construir la imagen virtual de Jikes RVM, el segundo parámetro indica si se va a incluir el compilador optimizado en el sistema y el tercer parámetro indica que esquema de recolección de basura de los disponibles para esta máquina virtual vamos a utilizar.
   En nuestro caso hemos elegido la configuración *jconfigure BaseBaseCopyMS*. Esto significa que la imagen de la máquina virtual será la básica (la máquina se va a construir con el compilador básico), así como que las clases cargadas y compiladas a instrucciones máquina en tiempo de ejecución también serán compiladas con el compilador básico y que el recolector de basura que vamos a utilizar es el *CopyMS*.

   El colector de basura *CopyMS* se caracteriza por ser un colector con un espacio de memoria utilizado para copia (en el cual se produce el alojamiento de memoria) y un espacio de memoria para elementos no copiados donde van los objetos supervivientes.

8. Ejecutar dentro de la carpeta **RVM_BUILD** la instrucción *./jbuild* que construye la máquina virtual Jikes RVM.
   La imagen de carga de Jikes RVM y los demás archivos generados durante el proceso de configuración y construcción de la máquina virtual son almacenados en el directorio rvmBuild creado previamente, con lo que está separado del directorio de trabajo, es decir del código fuente que se encuentra en otra carpeta.

ANEXO C: Relación entre la máquina virtual y el sistema operativo

Esta sección proporciona información sobre *"magic",* que es la forma que tiene Jikes RVM de solucionar el problema que se le presenta a la hora de implementar las funcionalidades que no pueden realizarse usando el lenguaje de programación Java. Por ejemplo, los recolectores de basura y el sistema de ejecución, en ocasiones, acceden a memoria y esto no es posible hacerlo directamente en Java. Sin embargo Jikes RVM no utiliza este código si no es absolutamente necesario.

Entre las funcionalidades que se necesita proporcionar podemos destacar la llamada a servicios del sistema operativo, el acceso a registros del procesador, el uso de instrucciones máquina específicas de la arquitectura sobre la que se esté ejecutando o la transferencia del flujo de ejecución a una dirección de memoria arbitraria.

Existen tres partes "mágicas" que describiremos a lo largo de esta sección.

- La primera parte es una colección de métodos "mágicos" que son métodos estáticos de la clase *VM_Magic* y que sirven para implementar las llamadas al sistema necesarias para el desarrollo de Jikes RVM.

- La segunda son las clases "mágicas" *VM_Address*, *VM_Word, VM_Offset* y *VM_Extent* que se usan en partes del paquete "*runtime"* y del recolector de basura.

- La tercera son varios mecanismos para declarar código ininterrumpible.

La clase *VM_Magic*

Algunos métodos de la clase *VM_Magic* son tratados de forma diferente por el compilador. Esto es debido a que algunos de estos métodos acceden a la memoria física y a la máquina de estados del procesador debido a que implementan llamadas al sistema operativo, y por lo tanto no pueden ser implementadas en el lenguaje de programación Java. Un desarrollador de Jikes RVM debe ser muy cauteloso a la hora de escribir código que utiliza estas clases para poder sortear el sistema de tipos que proporciona el lenguaje de programación Java.

Hay un método especialmente característico de esta clase y que se utiliza mucho que es el *VM_Magic.objectAsAddress* y el *VM_Magic.addressAsObject* que pasándole un objeto como parámetro de entrada te devuelve la dirección, como *VM_Address*, donde está ubicado el objeto en memoria, y viceversa.

Este método lo hemos utilizado en repetidas ocasiones en el proyecto para comprobar que los objetos se guardaban en el heap de memoria que nosotros esperábamos. Por ejemplo, con este método nos hemos asegurado de que los hilos principales de nuestras aplicaciones, así como las pilas asociadas a esos

hilos estaban guardas en los nuevos espacios de memoria que habíamos diseñado, con fin de tener separadas unas aplicaciones de otras.

Todos los usos de los métodos proporcionados por *VM_Magic* deben ocurrir en métodos ininterrumpibles o entre llamadas a *VM.disableGC* (que deshabilita el colector de basura) y *VM.enableGC* (que activa el colector de basura). De esta manera se garantiza que todas las llamadas al método *VM_Magic.objectAsAddress* se ha llevado a cabo de forma satisfactoria y no se ha producido durante una recolección de basura, lo que puede provocar que el método devuelva *null*.

Debido a que *Magic* es inexpresable en el lenguaje de programación Java, no es sorprendente que los cuerpos de los métodos de *VM_Magic* estén vacíos o indefinidos. Por el contrario, para cada uno de esos métodos, las instrucciones Java para generar el código de estos métodos están cargadas en *OPT_GenerateMagic*, *OPT_GenerateMachineSpecificMagic* y en *VM_Compiler.*

De esta manera, cuando el compilador se encuentra con una llamada a uno de estos métodos "mágicos", en lugar de realizar una compilación normal, directamente inserta el código apropiado para dicho método dentro del método que lo invoca.

De todas formas, por seguridad, el cuerpo de estos métodos no está vacío del todo. Contienen código java para mostrar un mensaje de advertencia por pantalla. Así, en caso de haber algún problema, sabríamos que el fallo ha sido a la hora de compilar la máquina. El compilador no permite utilizar estos métodos en el código de las aplicaciones; sólo permite su uso en el código de la máquina.

Otro aspecto a tener en cuenta a la hora de usar estos métodos (más concretamente, a la hora de trabajar con direcciones de memoria física directamente) es la problemática que plantea el sistema recolector de basura. Cuando un recolector de basura emplea una técnica basada en copia (como *copying collection* o *generational collector*), dicho recolector se encarga de actualizar las referencias de los objetos copiados, pero las direcciones físicas que hubiese almacenadas en un momento dado no se actualizan con la nueva ubicación de los objetos. Esto nos puede llevar a manejar de forma errónea una zona de memoria.

Para evitar este problema, el hilo que vaya a realizar acciones sensibles a una recolección debe deshabilitar la recolección de basura, obteniendo así una "ventana de seguridad" hasta que dicho hilo termine esas acciones y vuelva a habilitar el sistema recolector de basura.

Deshabilitar el recolector de basura conlleva el que el hilo no puede crear nuevos objetos (no puede hacer ningún "*new*") pues en caso de hacerlo y no haber suficiente memoria, el sistema se colgaría. Hay que recalcar que el resto de hilos sí pueden realizar peticiones de memoria, pero en caso de necesitar liberar memoria, estos hilos quedarían bloqueados esperando a que el hilo que deshabilitó el recolector de basura vuelva a habilitarlo. En ese caso el recolector se lanzaría nada más ser habilitado de nuevo.

Ahora bien, las repercusiones de deshabilitar el recolector de basura son más profundas:

- Es necesario evitar realizar la carga de una clase pues eso incluye la petición de memoria, sin hacer ningún "*new*" explícitamente.

- Tampoco se puede realizar enlace dinámico ni conversión de tipos, pues estas operaciones podrían implicar realizar la carga de una clase.

- El hilo que ha deshabilitado la recolección de basura no debe entrar en un monitor, variable condición o semáforo que pueda estar controlado por otro hilo el cual pueda estar esperando a que se libere memoria. De darse esta situación nos encontraríamos en un interbloqueo (*deadlock)*.

Todo esto pone de manifiesto la necesidad que ya hemos comentado de utilizar estos métodos con mucha precaución.

De las aproximadamente 650 clases que componen el sistema Jikes, sólo unas 110 emplean métodos de la clase *Magic*, de las cuales solo 12 necesitan inhabilitar la recolección de basura.

La clase *VM_Address*

El tipo *VM_Address* se utiliza en Jikes RVM para representar una dirección dependiente de la máquina virtual. Inicialmente, el tipo básico empleado para representar las direcciones era los enteros pero esta aproximación era deficiente. Primero porque la abstracción que se hacía de las direcciones era insuficiente y segundo porque el tipo entero de Java esta representado con signo mientras que las direcciones es mucho más apropiado considerarlas sin signo. Además, la diferencia entre las comparaciones sin signo o con signo en enteros es inexpresable en el lenguaje de programación Java.

Para solucionar todos estos problemas, se utilizan las instancias de *VM_Address* para representar las direcciones en Jikes RVM. Además, esta clase soporta operaciones tan costosas como añadir un desplazamiento entero a una dirección para obtener otra dirección, calcular la diferencia entre dos direcciones y comparar direcciones. También existen dos métodos que requieren una atención especial: convertir una dirección en un entero y viceversa. Estos métodos deben ser evitados siempre que sea posible.

Mientras que una dirección no es realmente un objeto, debemos tener en cuenta lo siguiente:

- No hay que pasar una instancia de *VM_Address* cuando se espera un objeto.

- No sincronizar una instancia de *VM_Address.*

- No se pueden construir arrays de instancias de *VM_Address* sino que hay que crear un *VM_AddressArray*.

ANEXO D: Tablas de gráficas de rendimiento

Consumo de tiempo (en segundos)

		Real	Usuario	Sistema
Máquina original	Jess	3,794	2,599	1,205
	Raytrace	11,557	11,245	0,344
	Javac	1,294	1,15	0,15
	MPEGAudio	2,652	2,504	0,156
	MTRT	9,413	9,147	0,256
	Jack	13,117	11,665	1,471
Máquina original lanzando dos aplicaciones	Jess	7,588	5,198	2,41
	Raytrace	23,114	22,49	0,688
	Javac	2,588	2,3	0,3
	MPEGAudio	5,304	5,008	0,312
	MTRT	18,826	18,294	0,512
	Jack	26,234	23,33	2,942
Nuestra máquina	Jess	3,83	2,576	1,263
	Raytrace	11,641	11,237	0,42
	Javac	1,382	1,214	0,17
	MPEGAudio	2,673	2,53	0,149
	MTRT	9,494	9,056	0,45
	Jack	13,949	12,306	1,666
Nuestra máquina lanzando dos aplicaciones	Jess	8,11	5,817	2,287
	Raytrace	23,239	22,472	0,762
	Javac	2,158	1,95	0,215
	MPEGAudio	4,996	4,787	0,195
	MTRT	19,901	19,29	0,656
	Jack	24,11	21,447	2,652

Figura D-1: Tabla de consumo de tiempo.

Consumo de memoria (en páginas de 4 KB)

		Total	meta	imm	nur-vm	ms-vm	los-vm	nur1Ap1	ms1Ap1	los1Ap1	nur2Ap2	ms2Ap2	los2Ap2
Máquina original	Jess	1778	0	1056	344	0	378	0	0	0	0	0	0
	Raytrace	3194	0	1056	432	1120	586	0	0	0	0	0	0
	Javac	1778	0	1056	344	0	378	0	0	0	0	0	0
	MPEGAudio	1778	0	1056	344	0	378	0	0	0	0	0	0
	MTRT	13422	0	1056	10072	1072	1222	0	0	0	0	0	0
	Jack	2144	0	1056	0	672	416	0	0	0	0	0	0
Máquina original lanzando dos aplicaciones	Jess	3556	0	2112	688	0	756	0	0	0	0	0	0
	Raytrace	6388	0	2112	864	2240	1172	0	0	0	0	0	0
	Javac	3556	0	2112	688	0	756	0	0	0	0	0	0
	MPEGAudio	3556	0	2112	688	0	756	0	0	0	0	0	0
	MTRT	26844	0	2112	20144	2144	2444	0	0	0	0	0	0
	Jack	4288	0	2112	0	1344	832	0	0	0	0	0	0
Nuestra máquina	Jess	2108	0	1032	8	0	0	0	0	0	0	0	0
	Raytrace	3275	0	1056	8	256	276	472	896	311	0	0	0
	Javac	1778	0	1056	344	0	378	0	0	0	0	0	0
	MPEGAudio	1778	0	1056	344	0	378	0	0	0	0	0	0
	MTRT	13487	0	1056	8	256	354	10080	864	869	0	0	0
	Jack	2193	0	1056	0	256	276	0	464	141	0	0	0
Nuestra máquina lanzando dos aplicaciones	Jess	2958	0	1064	0	256	422	0	512	323	0	304	77
	Raytrace	2980	0	1056	0	256	344	0	464	311	0	256	293
	Javac	1846	0	1056	344	0	446	0	0	0	0	0	0
	MPEGAudio	1846	0	1056	344	0	446	0	0	0	0	0	0
	MTRT	8548	0	1056	8	256	344	2960	432	311	2448	224	509
	Jack	2751	0	1056	0	256	344	0	384	106	0	464	141

Figura D-2: Tabla de consumo de memoria.

Recolecciones de basura realizadas durante la ejecución de la aplicación.

		Número de recolecciones
Máquina original	Jess	0
	Raytrace	2
	Javac	0
	MPEGAudio	0
	MTRT	1
	Jack	3
Máquina original lanzando dos aplicaciones	Jess	0
	Raytrace	4
	Javac	0
	MPEGAudio	0
	MTRT	2
	Jack	6
Nuestra máquina	Jess	0
	Raytrace	2
	Javac	0
	MPEGAudio	0
	MTRT	1
	Jack	3
Nuestra máquina lanzando dos aplicaciones	Jess	1
	Raytrace	2
	Javac	0
	MPEGAudio	0
	MTRT	3
	Jack	6

Figura D-3: Tabla de recolecciones de basura

Bibliografía

- http://jikesrvm.sourceforge.net
- http://www.spec.org/jvm98/jvm98/doc/benchmarks/index.html
- "*The Jalapeño Virtual Machine*", Alpern et Al., IBM SYSTEMS JOURNAL Nº 2000
- "*Optimización dinámica para recolectores generacionales*", José Manuel Velasco, Katzalin Olcoz et COL. XIV JORNADAS DE PARALELISMO— LEGANÉS (MADRID), SEPTIEMBRE 2003
- The Design and Implementation of the Jalapeño Research VM for Java International Conference on Parallel Architectures and Compilation Techniques, September 9, 2001 (PACT01 Tutorial)
- "*The Jikes Research Virtual Machine Project: Building and open-source research community",* Alpern et Al., IBM SYSTEMS JOURNAL VOL. 4 Nº 2, 2005
- "*Myths and Realities: The performance Impact of Gargabe Collection",* Stephen M Blackburn, Perry Cheng, Kathryn S McKinley

Palabras clave

- JikesRVM
- Máquina Virtual
- Jalapeño
- Multi-aplicación
- Java
- JTOC
- Gestor de memoria
- JMTK
- CopyMS

Concesión de derechos

Los abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria como la documentación y/o el prototipo desarrollado.

Beatriz Rodríguez Jiménez	Marta Texidor Méndez de Vigo	David Viñas Domínguez

En Madrid, a 7 de Julio del año 2006.

Agradecimientos

Queremos agradecer su ayuda, especialmente, a Manel Velasco, por toda la información que nos ha proporcionado y por la paciencia con la que nos ha ayudado a resolver nuestros problemas.

A Katzalin Olcoz, nuestra directora del proyecto, por su amabilidad y por atendernos en cualquier momento para resolver las dudas que nos iban surgiendo.

A todos los que nos han soportado hablando de máquinas virtuales durante todos estos meses y a los que se han interesado por nuestro trabajo.